TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VILLE KUOPPALA

PILVIPALVELUN LIITTÄMINEN IBM WEBSPHERE COMMERCE –VERKKOKAUPPA-ALUSTAN KÄYTTÖLIITTYMÄÄN

Diplomityö

Tarkastaja: apulaisprof. Petri Ihantola
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 9.11.2016

TIIVISTELMÄ

VILLE KUOPPALA: Pilvipalvelun liittäminen IBM WebSphere Commerce -verkkokauppa-alustan käyttöliittymään
Tampereen teknillinen yliopisto
Diplomityö, 48 sivua, 2 liitesivua
Tammikuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastaja: apulaisprof. Petri Ihantola

Avainsanat: IBM WebSphere Commerce, pilvipalvelu, verkkokauppa

IBM WebSphere Commerce on verkkokauppa-alusta, joka toteuttaa hyvin paljon erilaista liiketoimintalogiikkaa yhdessä sovelluksessa. Tämän seurauksena IBM WebSphere Commerce on sovelluskehittäjille hankalasti ylläpidettävä ja kehitettävä monoliittinen järjestelmä. IBM WebSphere Commercea on myös kehitetty pitkään verkkokauppa-alustana, minkä seurauksena kehitystyö on riippuvaista osittain vanhentuneista teknologioista.

Tässä työssä etsitään ratkaisua IBM WebSphere Commerce -verkkokauppa-alustan käyttöliittymän kehitystyön ongelmiin. Lisäksi työssä tutkitaan, olisiko verkkokaupan käyttöliittymäkehitystä mahdollista tehdä riippumatta IBM WebSphere Commercesta ja sen teknologioista.

Ratkaisuna IBM WebSphere Commercen aiheuttamiin ongelmiin esitetään pilvipalveluiden liittäminen verkkokaupan käyttöliittymätasolle. Tässä työssä verkkokaupan käyttöliittymään liitettävät pilvipalvelut toteuttavat IBM WebSphere Commercen käyttöliittymäkomponentteja. Pilvipalvelut toteutetaan mikropalveluarkkitehtuuria noudattaen, jolloin yksittäinen pilvipalvelu on pienehkö kokonaisuus, joka huolehtii omasta toiminnastaan. Mikropalveluarkkitehtuurin ansiosta palveluiden riippuvuudet muihin komponentteihin vähenevät ja yksittäisten mikropalveluna toteutettujen käyttöliittymäkomponenttien julkaisuprosessi helpottuu.

Tässä työssä toteutetun ratkaisun avulla osa käyttöliittymän kehitystyöstä voidaan tehdä riippumatta IBM WebSphere Commercesta ja sen teknologioista. Kehitettävän pilvipalvelun ei tarvitse noudattaa IBM WebSphere Commercen julkaisusykliä ja pilvipalvelun kehitys on helpompaa ja nopeampaa.

ABSTRACT

Ville Kuoppala: Adopting Cloud service into IBM WebSphere Commerce e-commerce platform UI
Tampere University of Technology
Master of Science Thesis, 48 pages, 2 appendix pages
January 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Assistant Professor Petri Ihantola

Keywords: IBM WebSphere Commerce, cloud service, e-commerce

IBM WebSphere Commerce is an e-com platform that implements a large amount of different business logic in one application. This leads to a monolithic software that is difficult to maintain and develop. IBM WebSphere Commerce has also been developed for a long time and due to this the development is dependent of partly legacy technologies.

In this thesis, a solution to the problems of developing the user interface of IBM WebSphere Commerce e-com platform is studied. Additionally, it is researched if the e-com user interface development is possible to do independently of IBM WebSphere Commerce and its technologies.

The solution to the problems is adopting cloud services into IBM WebSphere Commerce user interface. In this thesis cloud services that are attached to the e-com user interface are user interface components. Cloud services are implemented by following microservice architecture principles, which leads to small individual microservices that take independently care of their execution. Due to the microservice architecture there are less dependencies to other components and the user interface microservice components are easier to deploy.

The solution implemented in this thesis enables the development of the user interface to be independent of IBM WebSphere Commerce and its technologies. The cloud service that is developed does not need to follow the deployment cycle of IBM Web-Sphere Commerce and the development of cloud service is easier and faster.

ALKUSANAT

Tampereella 28. joulukuuta 2016

Tämä diplomityö on tehty Solteq Oyj:lle vuonna 2016.

Haluan kiittää apulaisprofessori Petri Ihantolaa työni asiantuntevasta ohjaamisesta ja työtä edistäneistä huomioista. Kiitos myös Solteq Oyj:n Valtteri Harmaiselle diplomityön aiheesta ja toteutuksen suunnittelusta kanssani. Erityiskiitos kannustuksesta ja työni oikolukemisesta Soile Kolehmalle.

Ville Kuoppala

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. WEBSOVELLUSARKKITEHTUURIT 4
2.1 Websovellukset yleisesti 4
2.2 Palvelupohjainen arkkitehtuuri websovelluksissa 6
2.3 Mikropalveluarkkitehtuuri websovelluksen laajentamisessa 10
2.4 Websovellusarkkitehtuurien vertailu 12
3. IBM WEBSPHERE COMMERCE 13
3.1 IBM WebSphere Commerce verkkokauppa-alustana 13
3.2 Widgetit IBM WebSphere Commercessa 14
3.3 IBM WebSphere Commercen hallintatyökalut 16
4. PILVIPALVELU-WIDGETIN TOTEUTUS 19
4.1 Arkkitehtuuri pilvipalvelu-widgetille 19
4.2 Pilvipalvelu-widgetin toteutus 21
4.3 Pilvipalvelu-widgetin käytön toteutus 23
4.3.1 Hallintatyökalutuen toteuttaminen pilvipalvelu-widgetille 23
4.3.2 ESpot-tuen toteuttaminen pilvipalvelu-widgetille 29
4.4 Pilvipalvelu-widgetin käyttö 34
4.4.1 Pilvipalvelu-widgetin käyttö hallintatyökalun kautta 34
4.4.2 Pilvipalvelu-widgetin käyttö eSpot-toiminnallisuutta hyödyntäen 35
4.5 IBM WebSphere Commerceen liitettävä pilvipalvelu 37
4.5.1 Rajoitteet pilvipalvelun toteutuksessa 37
4.5.2 Pilvipalvelun kommunikointi rajapinnoilla IBM WebSphere Commercen kanssa 38
4.6 Ratkaisun arviointi 40
5. YHTEENVETO JA JATKOKEHITYS 42
5.1 Yhteenveto 42
5.2 Pilvipalvelu-widgetin jatkokehitys 44
LÄHTEET 46

LIITE A: Pilvipalvelu-widgetin datan tarjoaja

LYHENTEET JA MERKINNÄT

WCS	WebSphere Commerce Suite on vaihtoehtoinen termi IBM WebSphere Commercelle
JSP	Java Servlet Page
JSPF	Java Servlet Page Fragment
Data bean	Data bean on Java bean, jonka avulla JSP-sivut voivat hakea ja näyttää dynaamista sisältöä
Pilvipalvelu	Tässä työssä pilvipalvelulla tarkoitetaan pilvipalvelua sekä pilvisovellusta
Pilvisovellus	Pilvipalvelussa ajettava pilvisovellus
Pilvipalvelu-widget	WCS:n widget, joka liittää pilvipalvelun käyttöliittymään
Widget	WCS:n yksittäinen käyttöliittymäkomponentti
Management Center	WCS:n hallintatyökalu yrityskäyttäjille
Commerce Composer	WCS:n hallintatyökalussa oleva käyttöliittymän muodostamiseen tarkoitettu työkalu
eSpot	Käytetään markkinointisisällön näyttämiseen verkkokaupan käyttöliittymässä

1. JOHDANTO

Nykypäivän websovelluskehitykselle ovat tyypillisiä jatkuvasti muuttuvat vaatimukset, tiukat aikataulut ja sovelluksen jatkuva laajentaminen. Näihin vaatimuksiin vastaaminen on haastavaa ja entistä vaikeampaa, jos taustalla käytetään useita vuosia kehitettyä monoliittista alustaa. Pitkään käytetyssä ja kehitetyssä alustassa on usein kokonaisuuksia, jotka käyttävät nykypäivänä vanhentuneita teknologioita. Tällöin websovelluksen jatkokehittäminen saattaa olla hidasta ja työlästä.

Tällaiset monoliittiset järjestelmät ovat verrattavissa legacy-järjestelmiin, jotka ovat vanhentuneilla teknologioilla toteutettuja, usein suuria, sovelluskokonaisuuksia. Muutosten tekeminen ja uusien ominaisuuksien toteuttaminen legacy-järjestelmään voi vaatia jopa niin paljon aikaa, että kokonaan uuden sovelluksen rakentaminen onnistuisi nopeammin. Koska muutosten tekeminen vie aikaa, ovat jatkuvasti muuttuvat vaatimukset suuri haaste, mikä taas johtaa aikataulujen venymiseen. Legacy-järjestelmän laajentuessa sen suorituskyky ei välttämättä enää riitä nykypäivän vaatimuksiin.

Legacy-järjestelmiä on kuitenkin vielä käytössä, koska ne sisältävät usein hyvin paljon jo olemassaolevaa dataa ja liiketoimintalogiikkaa. Data on saatettu kerätä yhteen eri järjestelmistä integraatioiden kautta, jolloin kaikki data on yhdessä järjestelmässä keskitetysti. Datan ja jo toteutetun liiketoimintalogiikan siirtäminen vanhasta järjestelmästä uuteen voi vaatia hyvin paljon työtä, minkä vuoksi joissain tilanteissa kannattaakin säilyttää vanha järjestelmä ja pyrkiä laajentamaan sitä moderneilla teknologioilla.

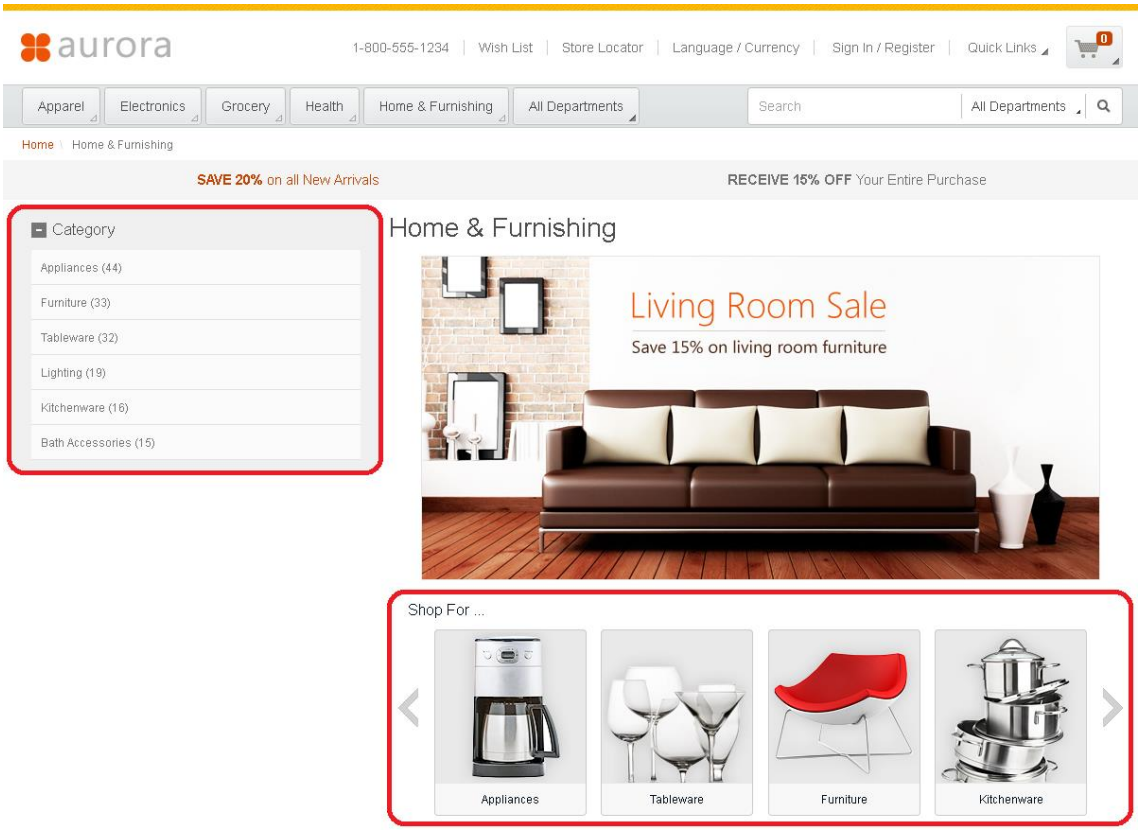
Tämän työn tarkoituksena on tutkia, kuinka websovellusta voidaan jatkokehittää ja laajentaa siten, että sovelluksen käyttöliittymän kehitys ei riippuisi käytettävästä alustasta ja sen teknologioista. Erityisesti tarkastellaan sitä, kuinka websovelluksen käyttöliittymäkomponentit olisivat luotavissa ulkoisiksi palveluiksi tai muutettavissa sellaisiksi.

Tässä työssä keskitytään IBM WebSphere Commerce –verkkokauppa-alustalla toteutetun verkkokaupan laajentamiseen pilvipalveluilla. Työssä käytetään myös vaihtoehtoista lyhennettä WCS (WebSphere Commerce Suite) IBM WebSphere Commerce -verkkokauppa-alustalle.

IBM WebSphere Commerce on verkkokauppa-alusta, jolla tehdään verkkokauppoja. IBM WebSphere Commercen verkkokaupan asiakas- ja palvelinpää ovat laajennettavissa ja kehitettävissä WCS:n tarjoamilla teknologioilla ja työkaluilla. IBM WebSphere Commerce ei varsinaisesti ole legacy-järjestelmä, sillä sitä kehitetään edelleen, mutta se on rinnastettavissa sellaiseen laajuutensa vuoksi. Työn tavoitteena onkin IBM WebSphere

Commercen käyttöliittymän kehitystyön tehostaminen siten, että käyttöliittymän kehitystyö olisi riippumatonta IBM WebSphere Commercesta ja sen teknologioista.

WCS:n käyttöliittymä muodostuu yksittäisistä käyttöliittymäkomponenteista, joita kutsutaan widgeteiksi. WCS:ssä yksi widget toteuttaa yhden käyttöliittymäkomponentin. Kuvassa 1 näkyy IBM WebSphere Commercen esimerkkiverkkokaupan widgetejä punaisella ympyröitynä.



Kuva 1: IBM WebSphere Commercen esimerkkiverkkokaupan itsenäisiä käyttöliittymäkomponentteja eli widgetejä

WCS mahdollistaa käyttöliittymäkomponenttien kehityksen itsenäisinä käyttöliittymäkomponentteina. Nämä widgetit ovat kuitenkin riippuvaisia WCS:n teknologioista ja WCS:n palvelinpään toteutuksesta, jolloin kehitystyössä käytettävät teknologiat rajoittuvat IBM WebSphere Commercen tarjoamiin teknologioihin. Tässä työssä pyritään laajentamaan verkkokauppaa itsenäisesti kehitettävillä käyttöliittymäkomponenteilla, jotka olisivat teknologiariippumattomia, jolloin kehitysteknologiat ja -työkalut olisivat vapaammin valittavissa.

Yksi mahdollisuus teknologiariippumattomien ja itsenäisten käyttöliittymäkomponenttien luontiin ovat pilvipalvelut. Perusajatus käyttöliittymän laajentamiseen pilvipalveluilla on luoda ulkoinen sovellus, joka liitetään taustajärjestelmän käyttöliittymään. Tällöin pilvipalvelu on kehitettävissä riippumatta toisesta järjestelmästä. Pilvipalvelu voi

kuitenkin tarvittaessa myös kommunikoida taustajärjestelmän kanssa rajapintoja käyttäen.

Tässä työssä toteutetaan IBM WebSphere Commerceen widget, johon voidaan tuoda käyttöliittymässä näytettävää ja toiminnallista sisältöä pilvipalvelusta. Työssä toteutettava widget tulee olemaan yleiskäyttöinen toisin kuin WCS:n tavalliset widgetit. Yleiskäyttöisyydellä tarkoitetaan sitä, että sama widget voi toteuttaa useita eri käyttöliittymäkomponentteja. Käytännössä toteutettu widget voidaan siis lisätä useita kertoja verkkokaupan käyttöliittymään siten, että se tuo käyttöliittymään useita eri ulkoisia palveluita. Lopulta toteutetun widgetin avulla on mahdollista liittää uusia tai korvata vanhoja käyttöliittymäkomponentteja verkkokaupan käyttöliittymässä.

Tämän työn toisessa luvussa käsitellään eri websovellusarkkitehtuureita sekä pohditaan niiden etuja ja ongelmia. Kolmas luku käsittelee IBM WebSphere Commercea verkkokauppa-alustana, WCS:n widgetejä sekä sen hallintatyökaluja. Neljännessä luvussa toteutetaan pilvipalvelun liittävä pilvipalvelu-widget WCS:ään. Samassa luvussa toteutetaan myös kaksi eri tapaa käyttää pilvipalvelu-widgetiä, esitellään eri toteutettujen tapojen käyttöä ja arvioidaan toteutettua ratkaisua. Viides luku on työn yhteenveto, jossa yhteenvedon lisäksi pohditaan työn jatkokehitysmahdollisuuksia.

2. WEBSOVELLUSARKKITEHTUURIT

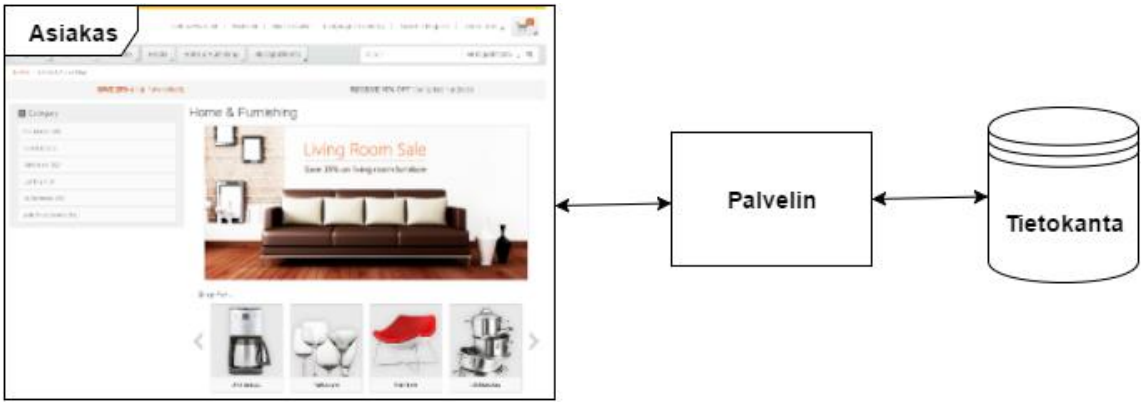
Tässä luvussa käsitellään websovellusarkkitehtuureita. Aluksi esitellään websovellukset yleisesti sekä tavallisimmat websovellusarkkitehtuurit eli asiakas-palvelin-arkkitehtuuri ja kolmikerrosarkkitehtuuri. Luvussa esitellään myös palvelupohjainen arkkitehtuuri sekä mikropalveluarkkitehtuuri ja lopuksi vertaillaan eri websovellusarkkitehtuureja keskenään.

2.1 Websovellukset yleisesti

Websovellukset ovat usein asiakas-palvelin-sovelluksia. Asiakas-palvelin-arkkitehtuuri koostuu tyypillisesti selaimesta eli asiakkaasta ja palvelimesta, joka sisältää usein suurimman osan sovelluksen liiketoimintalogiikasta. Asiakas-palvelin-malli toimii käytännössä siten, että asiakas tekee pyyntöjä palvelimelle ja palvelin käsittelee pyynnöt liiketoimintalogiikallaan ja vastaa asiakkaalle sovitussa muodossa. Palvelimen vastausten perusteella voidaan piirtää näkymä loppukäyttäjän selaimessa. [1]

Yksi esimerkki nykyaikaisesta asiakas-palvelin-sovelluksesta on verkkokauppa. Verkkokaupoissa käyttäjät voivat esimerkiksi selailla tuotekatalogeja, hakea tuotteita tai tehdä tilauksia. Palvelin sisältää usein liiketoimintalogiikan lisäksi erilaista tietoa, kuten esimerkiksi verkkokaupan tapauksessa tuote-, tilaus- ja varastotietoja. Lopulta loppukäyttäjä voi hakea verkkokaupan dataa, joka on käsitelty palvelinpään liiketoimintalogiikalla vastaamaan asiakaspään tarpeita. Esimerkki asiakaspään näkymästä on tuotesivu, joka hakee palvelimelta tuotteeseen liittyvän datan ja näyttää sen jäsennellyssä muodossa loppukäyttäjälle. [2]

Suurin osa websovelluksista, kuten myös verkkokaupat, voidaan usein jakaa arkkitehtuurinsa puolesta kolmeen eri kerrokseen: asiakkaaseen, palvelimeen ja tietokantaan [3]. Tätä jakoa kutsutaan kolmikerrosarkkitehtuuriksi tai useamman tietokannan tapauksessa monikerrosarkkitehtuuriksi. Kolmikerrosarkkitehtuuri on esitelty kuvassa 2, josta nähdään, kuinka asiakas keskustelee palvelimen kanssa ja palvelin taas tietokannan kanssa.

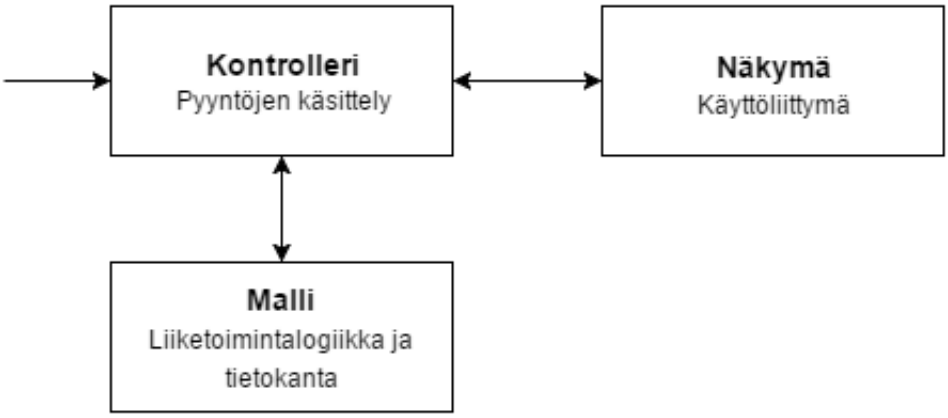


Kuva 2: Kolmikerroksisen websovelluksen arkkitehtuuri

Kolmikerrosarkkitehtuurin eri kerrokset ovat itsenäisiä kokonaisuuksia, jolloin eri kerroksia voidaan kehittää riippumatta toisista kerroksista. Yleisemmin kolmikerrosmalli jakautuu esityskerrokseen, loogiseen kerrokseen ja datakerrokseen. Esityskerros on websovellusten tapauksessa usein selain, joka piirtää loppukäyttäjälle verkkosivuston näkymän. Datakerros sisältää tietokannan datamallit ja on vastuussa datan tallentamisesta. Liiketoimintakerros eli palvelin sisältää websovelluksen liiketoimintalogiikan ja yhdistää esitys- ja datakerrokset käsittelemällä datakerrokselta hakemaansa dataa ja välittämällä sen lopulta esityskerrokselle. [4]

Esityskerros voi jossain tapauksissa sisältää suuren osan liiketoimintalogiikasta, jolloin sitä kutsutaan rikkaaksi asiakkaaksi (engl. rich client). Kun suurin osa liiketoimintalogiikasta on asiakaspäässä, ja selain vastaa sovelluksen suorituksesta, voidaan palvelin jättää toteutukseltaan kevyemmäksi. Vastakohta rikkaalle asiakkaalle on kevyt asiakas (engl. thin client), jolloin asiakaspäässä ei ole juurikaan liiketoimintalogiikkaa, vaan lähes kaikki logiikka sijaitsee palvelimella. Tämä saattaa joskus johtaa siihen, että palvelimesta tulee pullonkaula websovelluksen suorituskyvylle. [3] Usein websovellukset kuitenkin ovat yhtä aikaa molempia: esimerkiksi verkkokaupan tapauksessa tuotehaku voi olla raskasta palvelimelle, kun taas esimerkiksi toiminnallinen mainosbanneri voi tehdä suurimman osan laskennasta asiakaspäässä.

Usein kolmikerrosmallin toteutuksissa on käytössä MVC-arkkitehtuuri eli malli-näkymä-kontrolleri-arkkitehtuuri (engl. model-view-controller). Tässä arkkitehtuurissa malli sisältää liiketoiminnan ja tietokantaoperaatiot, näkymä esittää loppukäyttäjän näkymän ja kontrolleri käsittelee näkymältä saapuneet pyynnöt ja lopulta välittää tietoa näkymälle. [5] Kuvassa 3 esitellään MVC-arkkitehtuuri.



Kuva 3: MVC-arkkitehtuuri. Perustuu lähteeseen [5]

Perinteinen kolmikerrosmalli esimerkiksi MVC-arkkitehtuurilla toteutettuna saattaa kasvaa monoliittiseksi järjestelmäksi kehitystyön jatkuessa ja sovelluksen laajentuessa. Etuna kolmikerrosmallissa on se, että on vain yksi sovellus, jota tarvitsee hallita ja jonka on skaalauduttava. Kolmikerrosmalli vaatii kuitenkin sovelluksen laajentuessa hyvin suuren kehitystiimin ja jokainen muutos vaatii koko sovelluksen julkaisun. Tällöin pienikin muutos saattaa vaikuttaa useaan paikkaan ja aiheuttaa koko sovelluksen regressiotestauksen ennen sen viemistä tuotantoon. Usein monoliittisen kolmikerrosmallia noudattavan järjestelmän oppimiskäyrä voi olla sovelluskehittäjälle korkea ja siksi kehittäjien perehdyttäminen on hidasta. [3,6]

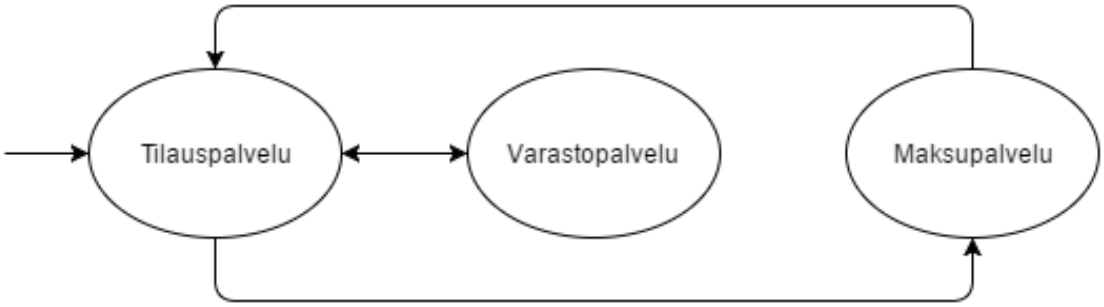
2.2 Palvelupohjainen arkkitehtuuri websovelluksissa

Monoliittiseksi kasvaneen kolmikerrossovelluksen ongelmia voidaan ratkaista jakamalla sovellus pienempiin itsenäisiin sovelluksiin. Tätä kutsutaan palvelupohjaiseksi arkkitehtuuriksi. Palvelupohjainen arkkitehtuuri muodostuukin palveluista, jotka ovat tämän työn tapauksessa monoliittisen järjestelmän osia pilkottuna pienemmiksi sovelluksiksi.

Palvelupohjainen arkkitehtuuri on arkkitehtuurityyli, joka perustuu liiketoimintalogiikkaa suorittaviin palveluihin. Palvelut kommunikoivat keskenään sovittujen rajapintojen kautta ja muodostavat yhdessä palvelupohjaisen järjestelmän. Tässä työssä palvelulla tarkoitetaan webpalvelua, joka suorittaa halutun toiminnallisuuden itsenäisesti omilla resursseillaan. Hyvä esimerkki webpalvelusta on pilvipalvelu, mutta myös muunlaiset palvelut ovat mahdollisia. Palvelut ovat käytettävissä yleensä verkon yli ja ovat itsenäisiä kokonaisuuksia, jolloin ne eivät ole teknologiariippuvaisia. Teknologiariippumattomuudella tarkoitetaan sitä, että palveluita voidaan kehittää millä tahansa teknologioilla tai työkaluilla, kunhan palvelun paljastama rajapinta pysyy samanlaisena.

Palvelupohjainen järjestelmä koostuu yksittäisistä palveluista, joita yhdistelemällä saadaan muodostettua korkeamman abstraktiotason liiketoimintalogiikkaa suorittava kokonaisuus [7]. Korkeamman abstraktiotason liiketoimintalogiikkaa voi olla esimerkiksi

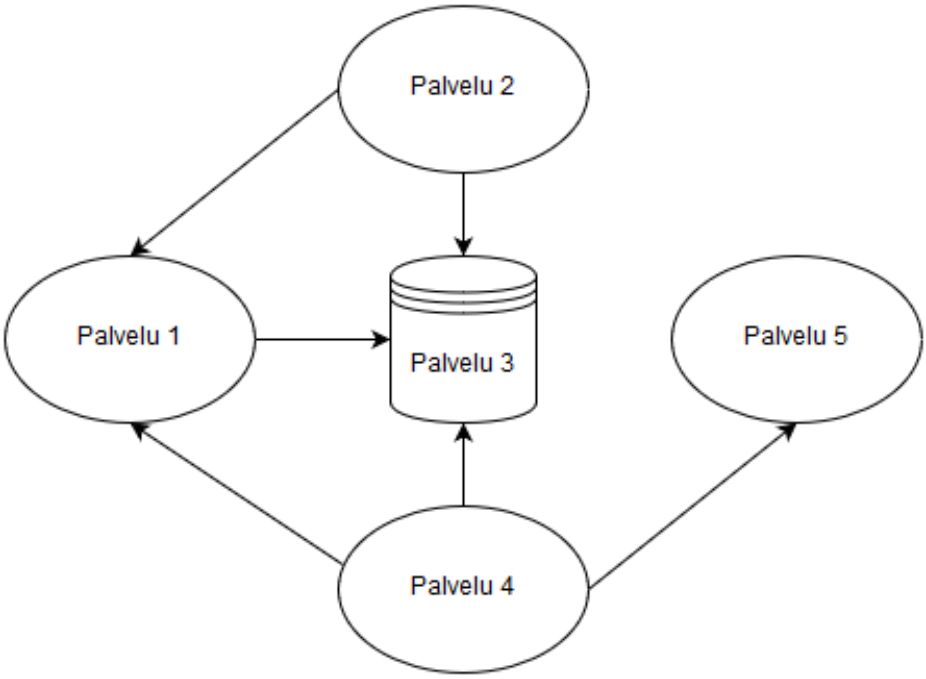
verkkokaupan tilausten hallinnan prosessi. IBM WebSphere Commercen tapauksessa tilausten hallinta on toteutettu yhtenä osana koko järjestelmää ja se voidaan jakaa karkeasti neljään osaan: tilauksen tallentamiseen, tilauksen prosessointiin, varaston prosessointiin ja maksun prosessointiin [8]. Kuvassa 4 esitetään sama toiminnallisuus yksinkertaistettuna palvelupohjaista arkkitehtuuria käyttäen, jossa tilausten hallinta on jaettu kolmeen palveluun.



Kuva 4: Verkkokaupan tilausprosessi palvelupohjaisena järjestelmänä, jossa nuolet esittävät sekä rajapintakutsuja että datan siirtymistä palveluiden välillä

Kuvan 4 tilauspalvelu kysyy varastopalvelulta (nuoli tilauspalvelusta varastopalveluun), ovatko tilauksen tuotteet saatavilla, jonka jälkeen se pyytää maksupalvelua suorittamaan maksun (nuoli tilauspalvelusta maksupalveluun). Lopulta maksupalvelu vastaa tilauspalvelulle maksun suorituksen onnistumisesta (nuoli maksupalvelusta tilauspalveluun), joka tallentaa tilauksen. Tilauspalvelu voi paljastaa rajapinnan tämän palvelupohjaisen järjestelmän ulkopuolelle, jolloin esimerkiksi verkkokauppa voi käyttää sitä tarvittaessa (nuoli tilauspalveluun).

Lopulta useista palveluista muodostuu esimerkiksi kuvan 5 kaltainen palvelupohjainen järjestelmä, jossa yksittäiset palvelut kommunikoivat toistensa kanssa toteutusriippumattomien rajapintojen kautta. Toteutusriippumattomuus palvelupohjaisten järjestelmien tapauksessa tarkoittaa sitä, että yksittäinen palvelu voidaan vaihtaa toiseen palveluun, joka voi olla eri teknologiolla toteutettu, kunhan rajapinnan kuvaus säilyy samana. [9]



Kuva 5: Palvelupohjainen järjestelmä, jossa palvelut ovat riippuvuvaisia palvelusta 3

Palvelupohjaiselle järjestelmälle tyypillistä on, että palveluita pyritään uudelleenkäyttämään mahdollisimman paljon. Esimerkki palveluiden uudelleenkäytöstä näkyy kuvassa 5, jossa lähes kaikki palvelut käyttävät keskimmäistä ”palvelu 3” -nimistä tietokantapalvelua. Palvelupohjainen arkkitehtuuri lupaa kehitystyöhön myös muita etuja kuten ketteryyttä, joustavuutta ja pienempiä kuluja. [7]

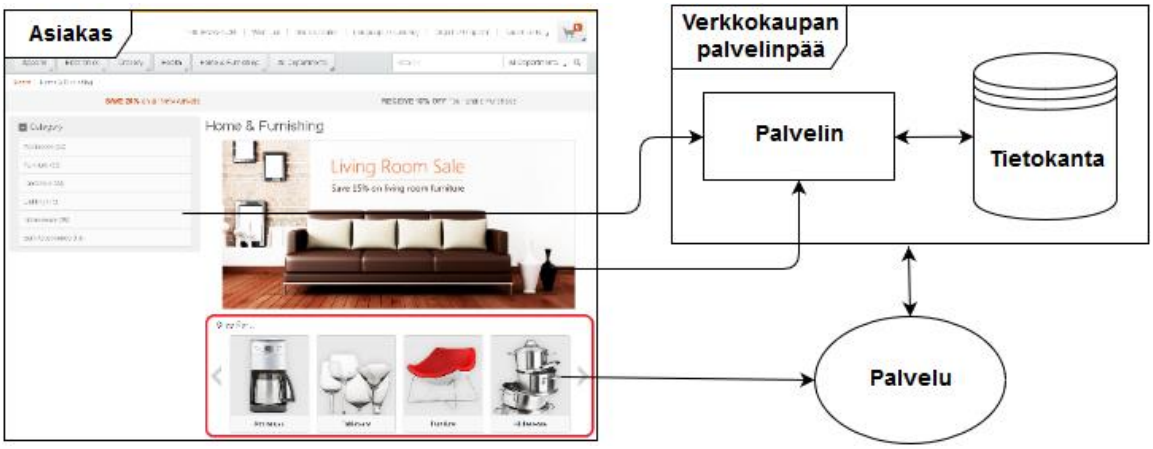
Yksi melko yleinen tapa palvelujen väliselle kommunikaatiolle ovat REST-rajapinnat. Käytännössä REST-rajapinnat perustuvat resursseihin, joita käsitellään HTTP-pyyntöjen metodeilla GET, POST, PUT ja DELETE.

HTTP-metodi	Resurssi	Seuraukset
GET	api/puu/koivu	Rajapinta palauttaa koivu-resurssiin liittyvän datan
POST	api/puu	Lisää uuden resurssin, joka voi olla esimerkiksi mänty
PUT	api/puu/mänty	Päivittää mänty-resurssin
DELETE	api/puu/koivu	Poistaa koivu-resurssin

Kuva 6: Esimerkki REST-rajapinnasta

Esimerkiksi puita käsittelevässä rajapinnassa api/puu/koivu on yksi resurssi, johon voidaan tehdä erilaisia pyyntöjä: hakea, lisätä, poistaa tai päivittää resurssia. Esimerkiksi

uuden resurssin lisääminen tapahtuu kohdistamalla POST-pyyntö resurssiin api/puu. Tällä tavoin on mahdollista lisätä rajapinta tarjoamaan esimerkiksi resurssi api/puu/mänty. Yksinkertainen puu-rajapinta, joka noudattaa REST-arkkitehtuurityyliä on esitelty kuvassa 6. [10]



Kuva 7: Tapa jakaa verkkokaupan käyttöliittymäkomponentteja palveluiksi

Yksi tapa laajentaa verkkokaupan käyttöliittymää on muuttaa yksittäisiä käyttöliittymäkomponentteja palveluiksi. Kuva 7 esittelee tavan jakaa verkkokaupan yksittäisiä käyttöliittymäkomponentteja palveluiksi palvelupohjaisen arkkitehtuurin periaatteita noudattaen. Kuvan 7 tapauksessa palvelu toteuttaa WCS:n käyttöliittymään tuotavan käyttöliittymäkomponentin ulkoasun ja toiminnallisuuden sekä kommunikoi tarvittaessa WCS:n palvelimen kanssa.

Palvelupohjainen arkkitehtuuri aiheuttaa kuitenkin joitakin rajoitteita itse palvelun toteuttamiselle. Rosen et al. [7] mukaan toteutettujen palveluiden tulisi olla samankokoisia, -muotoisia ja ylipäätään samankaltaisia toistensa kanssa, jotta palveluista tulee uudelleenkäytettäviä ja joustavia. Palveluiden tulisi myös kommunikoida yhdenmukaisella tavalla eivätkä niiden vastuualueet saisi olla päällekkäisiä. Palvelupohjaisessa arkkitehtuurissa pyritään käyttämään toteutettuja palveluja mahdollisimman paljon uudelleen, mikä aiheuttaa riippuvuuksia toisiin palveluihin, mistä seuraa ongelmia esimerkiksi palvelua julkaistaessa.

Koska palvelun kokoa ei ole määritelty, yksittäiset palvelut saattavat kasvaa hyvin laajoiksi järjestelmiksi, jolloin suurin osa palvelupohjaisen arkkitehtuurin eduista menetetään. Palveluiden suuri koko saattaa johtaa lopulta monoliittisiin palveluihin, jotka eivät enää ole helposti ylläpidettävissä. [11]

2.3 Mikropalveluarkkitehtuuri websovelluksen laajentamisessa

Mikropalveluarkkitehtuuri on palvelupohjainen arkkitehtuuri, jossa palvelut eivät ole vahvasti riippuvaisia toisista palveluista. Tässä arkkitehtuurissa palvelut ovat pieniin kokonaisuuksiin jaettuja palveluja ja mikropalveluarkkitehtuuria kutsutaankin usein hienojakoiseksi palvelupohjaiseksi järjestelmäksi (engl. fine-grained service-oriented architecture) [12]. Mikropalvelu nimityksenä on hieman harhaanjohtava, koska palvelut ovat yleensä kokonaisuuksia, jotka suorittavat jonkin osan liiketoimintalogiikasta. Toisin kuin palvelupohjaisessa arkkitehtuurissa, mikropalvelut eivät pyri yhtä voimakkaasti käyttämään uudelleen toisia palveluja, vaan toteuttavat ne usein itse, jolloin riippuvuudet muihin komponentteihin vähenevät. [11]

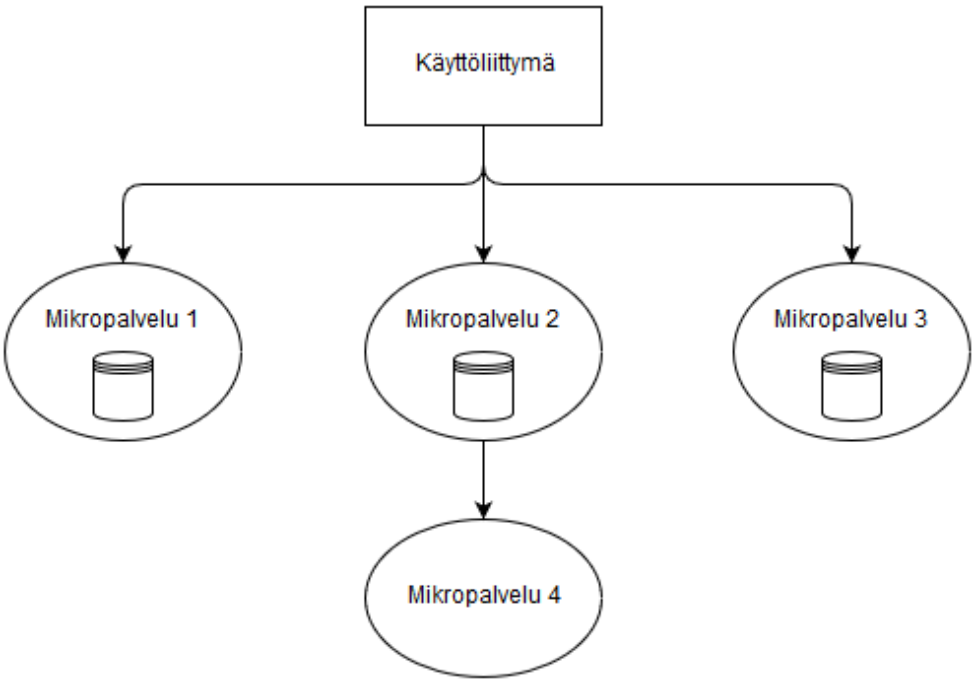
Mikropalveluarkkitehtuuri on vastakohta monoliittiselle järjestelmälle, jossa kaikki liiketoimintalogiikka on sijoitettu yhteen sovellukseen. Verrattuna suuriin järjestelmiin, joiden kanssa on haastavaa vastata nykypäivän liiketoiminnan ja asiakkaan luomiin vaatimuksiin, mikropalveluarkkitehtuurilla voidaan helpommin pilkkoa suuria kokonaisuuksia pienemmiksi ja näin tehostaa kehitystyötä. Mikropalveluarkkitehtuuri onkin luonteva valinta, kun tarvitaan tehokkuutta, jatkuvaa palvelua, luotettavuutta ja skaalautuvuutta. [13]

Verrattuna palvelupohjaisiin järjestelmiin mikropalveluihin muutosten tekeminen on helpompaa, koska vaikutus kohdistuu vain yhteen mikropalveluun eikä suurempiin kokonaisuuksiin. Vastaavasti mikropalvelujen ylläpitäminen on helpompaa, koska mikropalvelu ei riipu useista eri komponenteista tai palveluista, vaan sitä voidaan kehittää itsenäisesti. Koska julkaistavaa sovelluskoodia on mikropalveluissa vähemmän kuin tavallisissa palveluissa, pienenee todennäköisyys, että tehdyt muutokset vaikuttaisivat toisiin komponentteihin tai mikropalveluihin. [11]

Mikropalveluarkkitehtuuri perustuu usein pilvessä ajettaviin sovelluksiin, joita kutsutaan mikropalveluiksi. Pilvipalvelut skaalautuvat dynaamisesti eri kokoluokan tarpeisiin. Pilvipalvelumalli tarjoaakin liiketoimintamallin, jossa pilvipalvelun tarjoaja laskuttaa pilvessä ajettavista sovelluksista niiden käyttöasteen mukaan. Pilvipalvelujen käyttämisestä hyödytään siten, että sovellus on saatavilla jatkuvasti, se skaalautuu joustavasti ja kestää hyvin virheitä. [13]

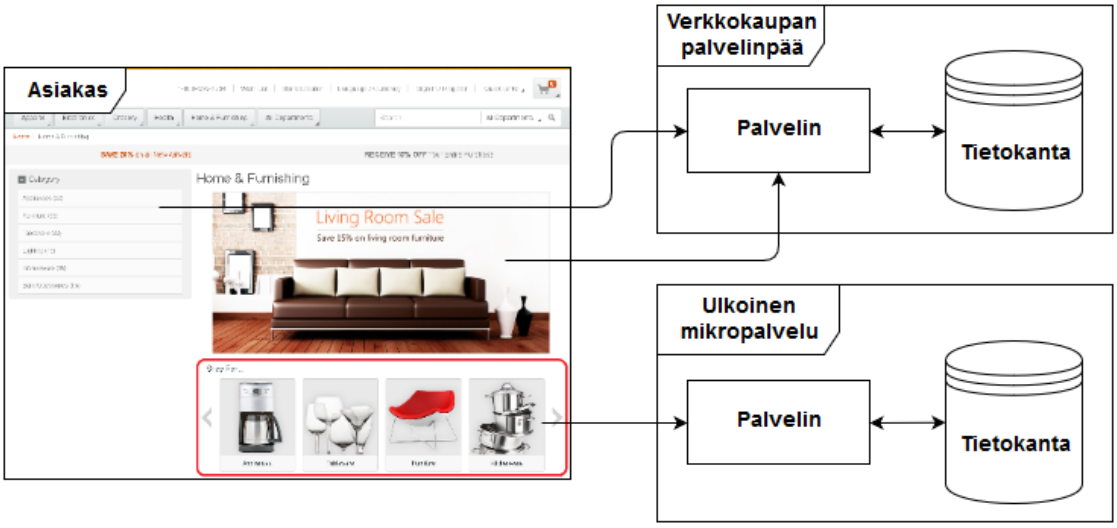
Verrattuna kuvan 5 palvelupohjaiseen arkkitehtuuriin kuvassa 8 esitetyssä mikropalveluarkkitehtuurissa jokainen mikropalvelu toteuttaa itsenäisesti toiminnallisuutensa eikä ole yhtä riippuvainen toisista komponenteista. Kuvan 8 tapauksessa mikropalvelut toteuttavat

koko käyttöliittymän, jolloin käyttöliittymästä tulee rikas asiakas, koska palvelujen käyttö tapahtuu pääosin käyttöliittymäsovelluksessa [3, 11].



Kuva 8: Mikropalveluarkkitehtuuri, jossa käyttöliittymä on koostettu mikropalveluista

Luotuja mikropalveluja voidaan kasata monoliittisen järjestelmän päälle, jotta saadaan vähennettyä suuremman järjestelmän kuormaa, jolloin uusia ominaisuuksia voidaan kehittää nopeammin ja helpommin nykyaikaisilla teknologioilla. Lopulta jopa koko monoliittinen ja hankalasti ylläpidettävä järjestelmä on tarvittaessa mahdollista korvata kokonaan mikropalveluilla kuvan 8 mukaisesti.



Kuva 9: Tapa jakaa verkkokaupan käyttöliittymäkomponentteja mikropalveluiksi

Kuvassa 9 esitetään tapa liittää mikropalveluarkkitehtuurin periaatteita noudattaen toteutettu käyttöliittymäkomponentti verkkokaupan käyttöliittymään. Ulkoinen pilvipalvelu

toteuttaakin kaiken toiminnallisuuden itse ja säilöö oman datansa, jolloin riippuvuudet muihin palveluihin minimoidaan. Tarvittaessa mikropalvelu voi kommunikoida verkkokaupan palvelimen kanssa rajapintoja käyttäen.

2.4 Websovellusarkkitehtuurien vertailu

Suurimpana erona palvelupohjaisen ja mikropalveluarkkitehtuurien välillä on palveluiden uudelleenkäyttö. Mikropalveluarkkitehtuuri pyrkii minimoimaan palvelujen uudelleenkäytön ja toteuttamaan tarvittavan toiminnallisuuden itsenäisesti, kun taas palvelupohjaisissa järjestelmissä tyypillisiä ovat vahvat riippuvuudet toisiin palveluihin. [11]

Mikropalveluarkkitehtuurissa ei yleensä ole yksittäistä palvelua, joka kasaa useita mikropalveluita yhden palvelun taakse, vaan usein esimerkiksi käyttöliittymä on vastuussa mikropalveluiden käytöstä. Palvelupohjaisissa järjestelmissä on usein yksittäinen keskitetty palvelu, joka kutsuu useita palveluita, joilla taas saattaa olla riippuvuuksia toisiin palveluihin.

Käyttöliittymäkehitystä ajatellen palvelupohjainen arkkitehtuuri ja mikropalveluarkkitehtuuri jakautuu selkeästi kahteen erilaiseen asiakaspäähän. Mikropalveluarkkitehtuurissa käyttöliittymä on usein rikas asiakas, joka suorittaa paljon liiketoimintalogiikkaa, koska se vastaa mikropalvelujen käytöstä ja orkestroinnista. Palvelupohjaisessa arkkitehtuurissa käyttöliittymä voi olla kevyt asiakas, joka kutsuu yhtä palvelua, joka taas taustalla hoitaa kaikkien muiden tarvittavien palveluiden kutsumisen. [11]

3. IBM WEBSPHERE COMMERCE

Tässä luvussa esitellään IBM WebSphere Commercea verkkokauppa-alustana sekä miten ja millaisilla teknologioilla WCS:ää kehitetään. Luvussa kuvaillaan myös, mitä ovat IBM WebSphere Commercen widgetit, millainen on WCS:n widget-arkkitehtuuri sekä tämän työn kannalta oleelliset WCS:n hallintatyökalut.

3.1 IBM WebSphere Commerce verkkokauppa-alustana

IBM WebSphere Commerce on verkkokauppa-alusta, joka on suunniteltu tukemaan yrityksen eri liiketoimintamalleja. IBM WebSphere Commerce lupaa tarjoavansa yhdellä alustalla monipuolisen ja yksilöidyn asiakaskokemuksen sekä skaalautuvansa minkä kokoiseen liiketoimintaan tahansa tukemalla sekä kuluttaja- että yritykseltä-yritykselle-myyntiä. IBM WebSphere Commerce sisältää verkkokaupan perusominaisuuksien lisäksi myös paljon valmiiksi asennettuja toimintoja, kuten yrityskäyttäjän hallintatyökalut, jotka mahdollistavat mainoskampanjoiden ja tarjouksien luomisen sekä tuotekatalogien hallitsemisen. [14]

Liiketoiminnan kannalta IBM WebSphere Commercen on suunniteltu toteuttavan kolme päätehtävää. Ensimmäinen päätehtävistä on monikanavaisen ostokokemuksen tehostaminen. IBM WebSphere Commerce pyrkii tarjoamaan personoidun monikanavaisen ostokokemuksen, joka on johdonmukainen kivijalkakaupasta verkkokauppaan. Toinen päätehtävä on asiakaskeskeinen käyttökokemus: WCS sisältää valmiit liiketoimintapalvelut, joilla voidaan luoda yksilöllisiä asiakaskokemuksia. Kolmantena päätehtävänä WCS pyrkii keskittymään sekä liiketoiminnan että kehitystyön synnyttämiin vaatimuksiin ja näin olemaan johtavassa asemassa verkkokauppaliiketoiminnassa. [14]

Tässä työssä käsiteltävä IBM WebSphere Commercen versio 7 ohjelmistopäivitys Feature Pack 8:lla käyttää Struts 1.1 -ohjelmistokehystä, joka on aikaisemmin ollut suosittu ohjelmistokehys Java-sovelluksille. Struts perustuu kappaleessa 2.1 esiteltyyn MVC-arkkitehtuuriin.

Kehittäjän näkökulmasta tämä tarkoittaa sitä, että asiakaspään kehittäminen rajoittuu JavaServer Pages (JSP) -teknologiaan. JSP on käytännössä HTML-laajennos, joka sallii Java-sovelluksien osien tai XML-tagien sisällyttämisen HTML-sivustoihin. JSP-sivustot muodostetaan palvelinpäässä ja lopulta välitetään näkymäkerrokselle. JSP-sivustojen tarkoitus on helpottaa asiakaspään käyttöliittymän dynaamisesti generoitujen sivustojen kehittämistä. [15] WCS tarjoaa myös asiakaspään asynkronisen JavaScript-toteutuksen, jonka avulla sivustoa voidaan päivittää dynaamisesti.

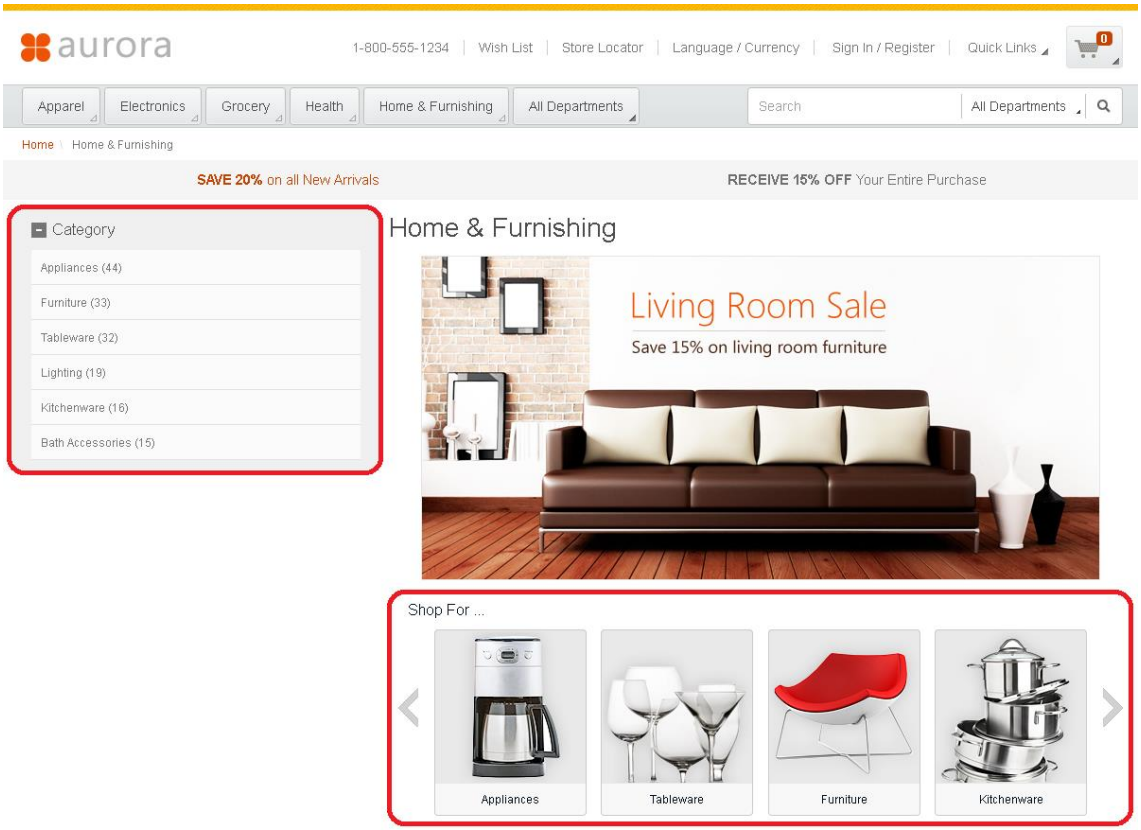
IBM WebSphere Commercen palvelinpäätä laajennetaan ja kustomoidaan Java EE -teknologioilla. WCS:n palvelinpää perustuu pitkälti EJB:n 1.1 – 2.X versioihin [16]. EJB eli Enterprise JavaBeans on arkkitehtuuri, joka koostuu Java-ohjelmointikielellä kirjoitetuista liiketoimintalogiikkaa sisältävistä komponenteista, joita suoritetaan palvelinpäässä. [17]

WCS:n käyttämät teknologiat siis rajoittavat palvelinpään kehityksen vanhentuneisiin teknologioihin kuten EJB 1.1 – 2.X. Palvelinpään kehityskieli on tämän seurauksena Java EE [16] [18]. Asiakaspään kehittäminen taas rajoittuu Struts 1.1 -ohjelmistokehyksen rajoituksiin ja JSP HTML-laajennoksen käyttöön [5]. Tämä aiheuttaa verkkokaupan käyttöliittymäkehitykseen ongelmia, sillä JSP-sivustot pohjautuvat vahvasti Java-teknologiaan, jolloin käyttöliittymäkehittäjän täytyy ymmärtää myös taustajärjestelmän toimintaa jollain tasolla.

3.2 Widgetit IBM WebSphere Commercessa

IBM WebSphere Commercen tapauksessa widgeteillä tarkoitetaan sivustolla esiintyviä käyttöliittymäkomponentteja. Widgetit ovat hallittavissa ja lisättävissä verkkokauppaan Management Center -hallintatyökalun Commerce Composer -työkalulla, jota tässä työssä tullaan kutsumaan pelkästään hallintatyökaluksi. Tällä hallintatyökalulla voidaan luoda ulkoasusuunnitelmia ja asettaa widgetejä ulkoasusuunnitelmiin. WCS:n verkkokaupan asiakaspää koostuukin kokoelmasta widgetejä, jotka on lisätty sivuston ulkoasusuunnitelmaan hallintatyökalussa.

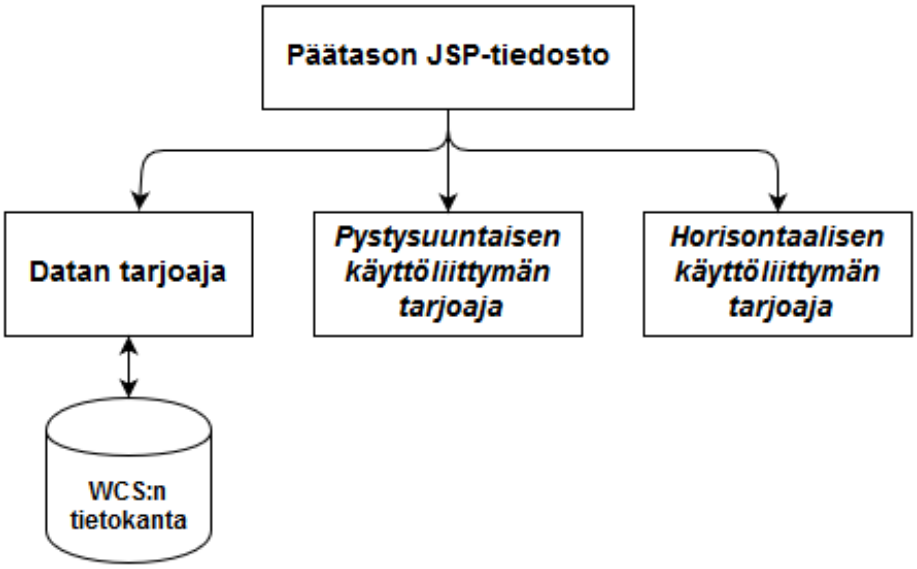
Jokainen widget on itsenäinen kokonaisuus, joka hakee ja näyttää widgetiin liittyvää dataa. Kuvassa 10 näkyy esimerkkiwidgetejä punaisella värillä ympyröitynä: vasemmassa reunassa on kategorioissa navigoimiseen tarkoitettu widget ja oikeassa alareunassa on sisältöä suositteleva kategoriakaruselli-widget. Esimerkiksi kategoriakaruselli-widgetin tapauksessa widget hakee palvelinpäästä eri kategoriat sekä niihin liittyvät metatiedot, piirtää widgetin käyttöliittymään ja suorittaa mahdollisesti siihen liitetyn JavaScript-logiikan. [19]



Kuva 10: IBM:n Aurora-esimerkkiverkkokaupan widgetejä ympyröitynä punaisella

IBM WebSphere Commercen widgetien toteutus koostuu kahdesta komponentista: käyttäjälle näkyvästä osuudesta, joka hakee sekä näyttää datan sivustolla eli itse widgetistä ja hallintatyökalun komponentista, jota käytetään widgetin hallitsemiseen kuten parametrien antamiseen widgetille. Käyttöliittymäkomponentti voi olla muokattu yrityksen tarpeisiin siten, että se näyttää eri sisältöä mahdollisesti eri tavoilla riippuen hallintatyökalun kautta tehdyistä konfiguraatioista. [19]

IBM WebSphere Commercen widget-arkkitehtuuri muodostuu kolmesta eri kerroksesta: päätason JSP-tiedostosta, alatason JSPF-tiedostoista (JavaServer Page Fragment) ja lopulta WCS:n tietokannasta, josta haetaan widgetin tarvitsemaa dataa. Kuvassa 11 esitellään WCS:n widget-arkkitehtuuri ja kuinka widgetin eri osat kommunikoivat keskenään. [19]



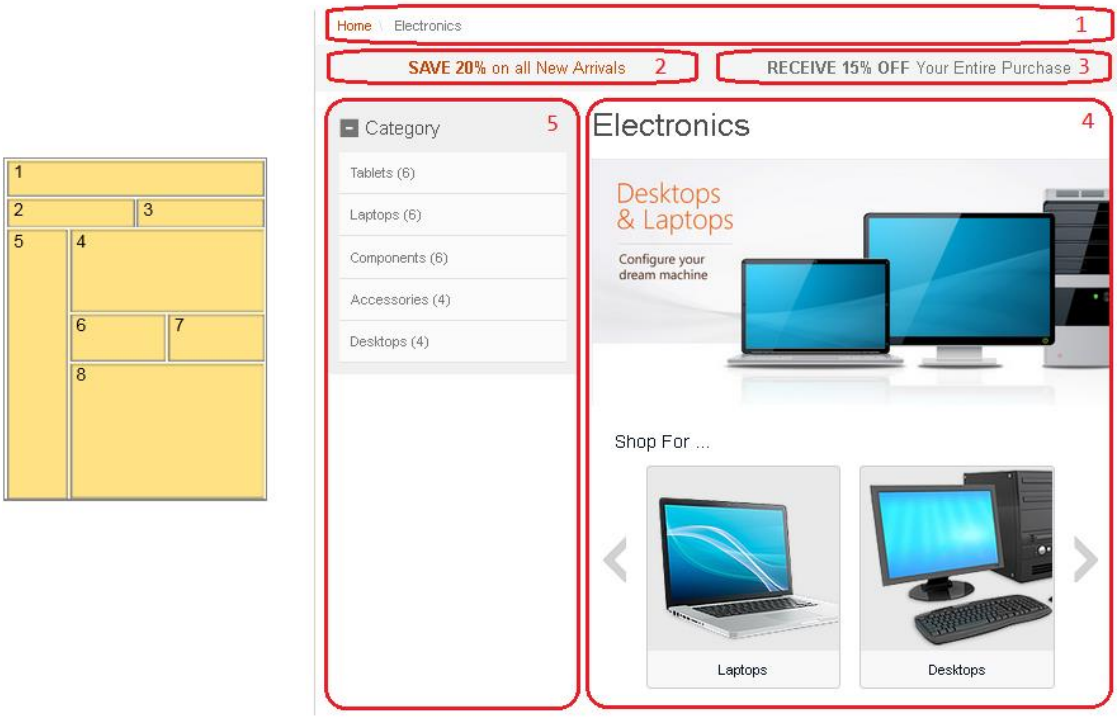
Kuva 11: IBM WebSphere Commercen widget-arkkitehtuuri

Widgetin JSP-sovelluskoodin suoritus alkaa kuvan 11 päätason JSP-tiedostosta. Päätason JSP-tiedosto kokoaa yhteen datan sekä valitsee käytettävän ulkoasun. Kuvan 11 esimerkkitapauksessa päätason JSP-tiedostossa valitaan joko horisontaalinen tai pystysuuntainen käyttöliittymä hallintatyökalussa asetettujen parametrien perusteella. Widgetillä voi olla useampia erilaisia ulkoasuja: esimerkiksi mobiilikäyttöliittymä tai käyttöliittymä, joka näytetään vain tietyllä sivulla tai tietylle käyttäjäryhmälle. Koko sivuston ulkoasusta on myös mahdollista tehdä skaalautuva eri laitteille, mikäli widgetiä kehitettäessä seurataan websovellussuunnittelumallia, joka tuottaa responsiivisia sivustoja. [19]

Widgetin datan tarjoajan tehtävänä on hakea tarvittava data WCS:n tietokannasta käyttäen Javan data beaneja, palvelupyyntöjä tai rajapintapyyntöjä. Lopulta päätason JSP-tiedosto välittää datan käytettävälle käyttöliittymän tarjoajalle ja widget piirretään sivustolle. Lisäksi widgetiä kehitettäessä voidaan tarvittaessa ottaa käyttöön CSS-tyyliohjeet ja JavaScript-ohjelmointikieli, jotka ovat yleisesti käytössä websovellusten asiakaspään kehityksessä ja suorituksessa. [19]

3.3 IBM WebSphere Commercen hallintatyökalut

Edellisessä kappaleessa esitellyt widgetit ovat hallittavissa ja lisättävissä käyttöliittymään IBM WebSphere Commercen mukana tulevasta Management Center -hallintatyökalusta. Tämä kuitenkin edellyttää tarvittavien määrittelytiedostojen luomista Management Centeriä varten [19]. Management Center koostuu useista eri osioista, mutta käyttöliittymä luodaan sen Commerce Composer -työkalulla. Kuvassa 12 esitellään Commerce Composerin ulkoasusuunnitelma ja sen kautta asetetut eri widgetit verkkokaupan käyttöliittymään.

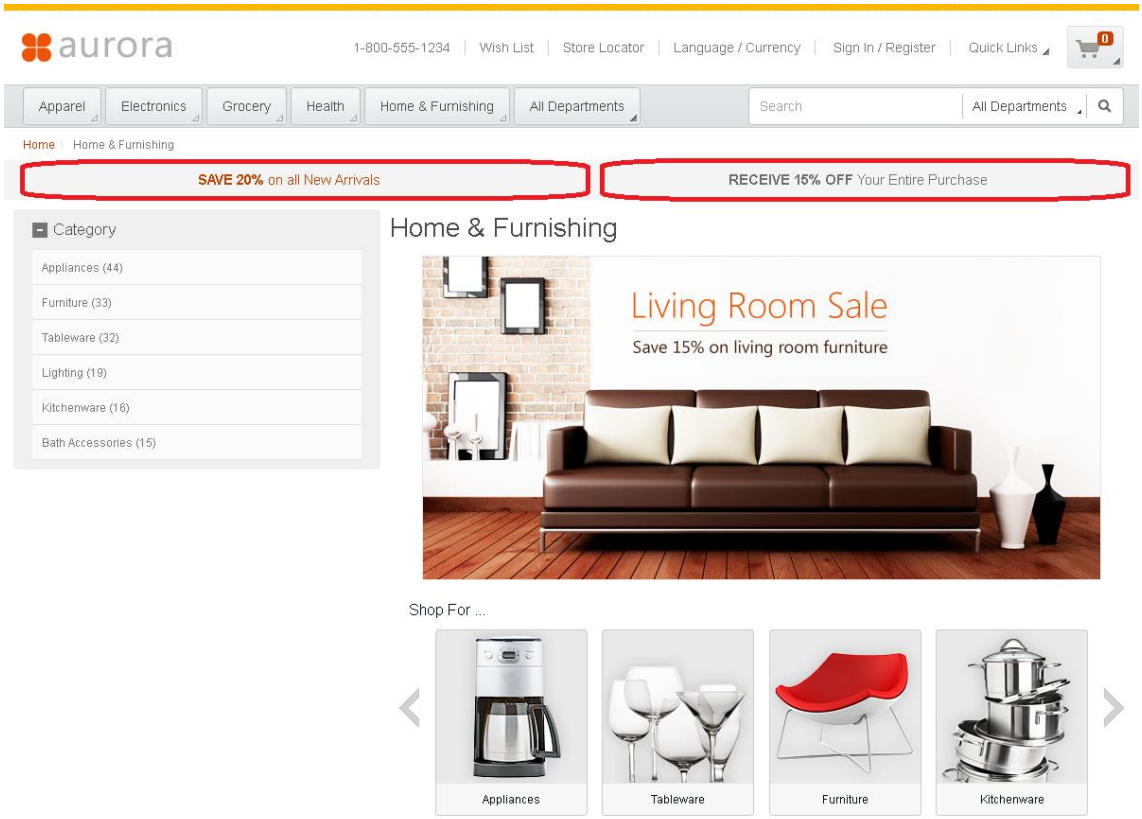


Kuva 12: Commerce Composerin ulkoasusuunnitelma ja toteutunut Aurora-esimerkkiverkkokaupan ulkoasu

Kuvasta 12 nähdään, että ulkoasusuunnitelman ennalta määritellyt paikat ovat usein suurempia kokonaisuuksia, kuten kategorialistauksia tai tuotekaruselleja otsikoineen. Commerce Composerin pääkäyttötarkoitus onkin luoda erilaisia ulkoasusuunnitelmia ja lisätä widgetejä eri ulkoasusuunnitelmien paikkoihin. Näistä ulkoasusuunnitelmista koostuu lopulta koko verkkokaupan käyttöliittymä.

Commerce Composeria käytettäessä widgetien lisääminen ei vaadi muutoksia sovelluskoodiin eikä palvelimen uudelleenkäynnistämistä muutosten käyttöönottamiseksi. Usein Commerce Composeriin on pääsy myös yrityksen edustajilla, jolloin verkkokaupan sisällöntuottajat voivat siirrellä ja järjestellä widgetejä eri järjestykseen helposti, eikä sovelluskehittäjiä tarvita muutosten tekemiseen.

IBM WebSphere Commerce tarjoaa myös e-marketing spot -nimisen (lyh. eSpot) konseptin, jotta sisällöntuottajat voisivat helposti vaihdella esimerkiksi erilaisia mainostekstejä verkkokaupan käyttöliittymässä. ESpotit ovat sijoiteltavissa manuaalisesti mihin tahansa kohtaan verkkokaupan käyttöliittymän JSP-tiedostoja. [20] Kuvassa 13 on kaksi eSpotia, joiden sisältö voidaan muuttaa tai asettaa halutuksi WCS:n hallintatyökalun markkinointiosiosta.



Kuva 13: Widgetejä, joihin tuodaan markkinointitekstit eSpot-toiminnallisuudelle

ESpot toimii lähes samalla tavalla kuin widget: se varaa tilan sivustolta ja näyttää yritys-käyttäjien määrittelemää markkinointisisältöä. Käyttämällä eSpoteja on mahdollista hallita ja muuttaa näytettävää sisältöä ilman muutoksia sovelluskoodiin ja julkaisematta sovellusta uudelleen. [20]

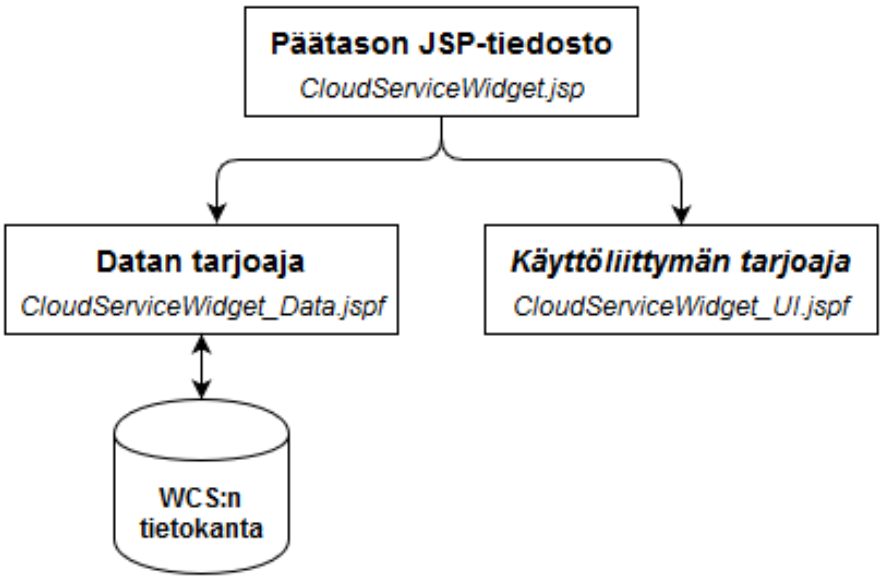
4. PILVIPALVELU-WIDGETIN TOTEUTUS

Tässä luvussa esitellään työssä toteutetun pilvipalvelu-widgetin tekninen toteutus sekä kuinka pilvipalvelu-widgetiä käytetään. Kappaleessa 4.1 esitellään arkkitehtuuri pilvipalvelu-widgetille. Kappaleessa 4.2 toteutetaan yksinkertainen arkkitehtuurin mukainen pilvipalvelu-widget, joka toimii tilanteissa, joissa kokonainen widget halutaan korvata pilvipalvelulla. Kappaleessa 4.3 laajennetaan pilvipalvelu-widgetin toteutusta siten, että pienempiä kokonaisuuksia, kuten WCS:n widgetin osia, voidaan korvata pilvipalvelulla. Luvun lopussa käsitellään pilvipalvelu-widgetin aiheuttamia rajoituksia pilvipalvelun toteutuksessa sekä kuinka pilvipalvelu-widget voi kommunikoida IBM WebSphere Commercen tai kolmansien osapuolien rajapintojen kanssa.

4.1 Arkkitehtuuri pilvipalvelu-widgetille

Tässä kappaleessa esitellään arkkitehtuuri pilvipalvelu-widgetille. Pilvipalvelu-widgetin tarkoituksena on hakea ja näyttää pilvipalvelu IBM WebSphere Commercen käyttöliittymätasolla. Pilvipalvelu-widgetin arkkitehtuuri noudattaa kappaleessa 3.2 esiteltyä WCS:n widget-arkkitehtuuria.

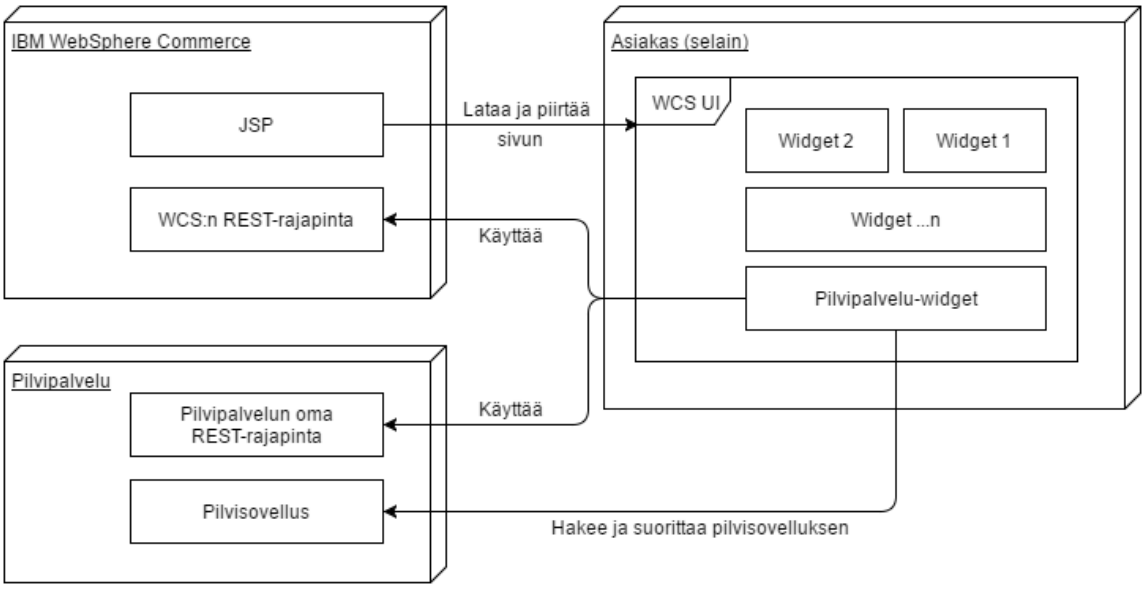
Pilvipalvelu liitetään verkkokauppaan tavallisella IBM WebSphere Commercen widgetillä, jonka käyttöliittymän tarjoaja hakee pilvisovelluksen verkkokaupan käyttöliittymätasolle. Pilvipalvelu-widgetin toteuttaminen käsitellään tarkemmin kappaleessa 4.2. Kuvassa 14 esitetään pilvipalvelu-widgetin arkkitehtuuri IBM WebSphere Commercen widget-arkkitehtuuria mukaillen.



Kuva 14: Pilvipalvelu-widgetin arkkitehtuuri

Kuvassa 14 näkyy tässä työssä toteutettavan widgetin tiedostojen nimet. Datan tarjoaja keskustelee WCS:n tietokannan kanssa hakeakseen widgetin vaatimia parametreja, joiden avulla tiedetään esimerkiksi pilvipalvelun sijainti. Käyttöliittymän tarjoajan tehtävänä on hakea ulkoinen pilvisovellus verkkokaupan käyttöliittymätasolle sekä piirtää ja suorittaa pilvipalvelu loppukäyttäjälle näkyvässä käyttöliittymässä. Tämän jälkeen käyttöliittymässä oleva pilvisovellus voi kommunikoida eri rajapintojen kanssa.

Kuvassa 15 esitetään pilvipalvelu-widgetin suhde IBM WebSphere Commercen palvelimeen ja toteuttavaan pilvipalveluun. Kuva 15 jakautuu asiakkaaseen, palvelimeen ja ulkoiseen palveluun, jotka tässä tapauksessa ovat selain, IBM WebSphere Commerce ja toteutettu pilvipalvelu. Kuvasta nähdään myös, kuinka pilvipalvelu-widget lopulta suoritetaan WCS:n asiakaspäässä.



Kuva 15: Pilvipalvelu-widgetin suhde toteutettavaan pilvipalveluun ja WCS:ään

Kuvan 15 oikeassa reunassa oleva selain toimii asiakkaan roolissa. Kuvassa 15 nähdään tavallisia widgetejä (kuvan widgetit 1-n) sekä tässä työssä toteutettava widgetin erikoistapaus pilvipalvelu-widget. Ensimmäisenä asiakasselain pyytää IBM WebSphere Commercen palvelimen valmistelemat JSP-tiedostot, joiden avulla selain osaa piirtää widgetit verkkokaupan käyttöliittymään. Samaa käytäntöä noudattaen myös tässä työssä toteutettava pilvipalvelu-widget haetaan verkkokaupan käyttöliittymään palvelimelta. Koska JSP-tiedostot suoritetaan palvelinpäässä, pilvipalvelu-widget hakee jo tässä vaiheessa tarvitsemansa parametrit hallintatyökalusta.

Kun pilvipalvelu-widget on selainpäässä, se hakee ulkoisesti tarjotun pilvisovelluksen annettujen hallintatyökaluparametrien perusteella. Pilvisovellus on toteutettu siten, että se osaa piirtää tarvittavat käyttöliittymäkomponentit WCS:n asiakaspäähän, suorittaa pil-

visovelluksen oman JavaScript-logiikan, sekä liittää sovellukseen tarvittavat tyylitiedostot. Lopulta pilvipalvelu-widgetin sisässä toimiva pilvisovellus voi kommunikoida joko WCS:n rajapintojen, pilvipalvelun oman rajapinnan kanssa tai mahdollisesti kolmansien osapuolien rajapintojen kanssa.

4.2 Pilvipalvelu-widgetin toteutus

Tässä kappaleessa toteutetaan pilvipalvelu-widget IBM WebSphere Commerce 7.0:aan, johon on asennettu Feature Pack 8 -päivityspaketti. Toteutetulla pilvipalvelu-widgetillä voidaan liittää pilvipalveluita IBM WebSphere Commercen käyttöliittymään.

Yleisellä tasolla widgetejä toteutetaan IBM WebSphere Commerceen luomalla aiemmin esitellyssä kuvassa 14 esiintyvät widget-arkkitehtuurin mukaiset JSP-tiedostot eli päätason tiedosto sekä alatason JSPF-tiedostot. Lisäksi widget täytyy määritellä vaadituissa tilaus ja rekisteröinti -tiedostoissa (engl. register, subscribe), jotta widget voidaan ottaa verkkokaupassa käyttöön. Tilaus ja rekisteröinti -tiedostot määritellään myöhemmin kappaleessa 4.3.1. [19]

Pilvipalvelu-widgetin toteutus aloitetaan luomalla päätason JSP-tiedosto. Widget-arkkitehtuurin mukaisesti tarvitaan vielä datan tarjoaja ja käyttöliittymän tarjoaja. Ohjelmassa 1 sisällytetään yksinkertaisesti alatason JSPF-tiedostot päätason JSP-tiedostoon.

```
1  <%@ include file="CloudServiceWidget_Data.jspf" %>
2  <%@ include file="CloudServiceWidget_UI.jspf" %>
```

Ohjelma 1: Päätason JSP-tiedosto CloudServiceWidget.jsp

Seuraavaksi luodaan pilvipalvelu-widgetille datan tarjoaja, joka esitellään ohjelmassa 2. Datan tarjoaja saa muuttujat containerParams ja jsParams param-muuttujan kautta. Parametrien välitystä hallintatyökalun kautta tullaan käsittelemään tarkemmin kappaleessa 4.3. Riveillä 1 ja 5 tarkastellaan, mikäli jommankumman parametrin arvo on jäänyt tyhjäksi, jonka jälkeen riveillä 2 ja 6 asetetaan param-muuttujan sisältö containerParams ja jsParams -nimisiin muuttujiin.

```
1  <c:if test="${!empty param.containerParams}">
2     <c:set var="containerParams" value="${param.containerParams}" />
3  </c:if>
4
5  <c:if test="${!empty param.jsParams}">
6     <c:set var="jsParams" value="${param.jsParams}" />
7  </c:if>
```

Ohjelma 2: Datan tarjoaja CloudServiceWidget_Data.jspf

Jäljellä on vielä käyttöliittymän tarjoajan toteutus, joka esitellään ohjelmassa 3. Käyttöliittymän tarjoaja on lopulta verkkokaupan käyttöliittymässä näkyvä osuus. Kyseinen toteutus aiheuttaa rajoituksia pilvipalvelun toteutukseen, joita käsitellään myöhemmin kappaleessa 4.5.1.

```
1  <c:if test="${!empty containerParams && !empty jsParams}">
2     <div ${containerParams}>
3        <!-- Pilvipalvelu liitetään tähän -->
4     </div>
5
6     <script type="text/javascript" ${jsParams}></script>
7  </c:if>
```

Ohjelma 3: Käyttöliittymän tarjoaja CloudServiceWidget_UI.jspf

Rivillä 1 ohjelma 3 varmistaa uudelleen, että containerParams ja jsParams -muuttujat eivät ole tyhjiä, jotta käyttöliittymäkerroksella voidaan näyttää sisältöä. Riveillä 2-4 näkyy div-lohko, jonka sisään pilvipalvelun HTML-rakenne liitetään esimerkiksi lohkon id:n perusteella. Ohjelma 3 sellaisenaan käyttöliittymään liitettynä ei näytä mitään loppukäyttäjälle, vaan lisää ainoastaan tyhjän div-lohkon ja script-tagin HTML-rakenteen joukkoon. Lohkon id välitetään containerParams-muuttujan kautta, jolloin containerParams:n arvo voi olla esimerkiksi id=”cloud-service-container”. Muuttuja containerParams on monikkomuodossa, jotta sen käyttötarkoitus ei nimen perusteella rajoittuisi ainoastaan yhteen parametriin. Joustavuuden ja mahdollisten tulevien vaatimusten huomioimiseksi containerParams:n kautta on mahdollista antaa div-lohkolle myös useampia parametreja, kuten luokkia tai muita div-lohkolle annettavia parametreja, id:n lisäksi.

Pilvisovelluksen käyttöliittymään hakemista varten rivillä 6 lisätään script-tagi, joka hakee suoritettavan JavaScript-sovelluksen käyttöliittymätasolle. Tässä työssä jsParams:n arvo voi olla esimerkiksi data-main=”https://esimerkki-url.com/app.js” src=”https://esimerkki-url.com/lib/require.min.js”. Jälkimmäisenä annettu src-parametri lataa require.js-JavaScript-kirjaston käyttöön, jota käytetään myöhemmin pilvipalvelun toteutuksessa. Käyttöön ladattu require.js-kirjasto osaa etsiä data-main-parametrina annetun ajettavan JavaScript-sovelluksen ja käynnistää sen suorituksen. [21]

Lopulta käynnistetty pilvisovellus sisällyttää div-lohkoon pilvipalvelun HTML-rakenteen, jonka se etsii containerParams-muuttujaan asetetun id:n perusteella. Myös jsParams-muuttuja on jätetty tarkoituksella vapaamuotoisesti määriteltäväksi, jotta script-tagin parametrit ovat joustavasti asetettavissa. Tämän jälkeen käyttöliittymäntarjoajaan muodostuu sovelluksen HTML-rakenne, jota app.js-sovellus voi käsitellä halutulla tavalla ja jonka kautta pilvisovellus saa syötettä. Lopullinen verkkokaupan käyttöliittymässä näkyvä HTML-rakenne esimerkkiparametreilla on esitelty ohjelmassa 4.

```
<div id="cloud-service-container">
   <!-- Ulkoisen esimerkkipilvipalvelun käyttöliittymän HTML-rakenne -->
</div>

<script type="text/javascript"
        data-main="https://esimerkki-url.com/app.js"
        src="https://esimerkki-url.com/lib/require.min.js">
</script>
```

Ohjelma 4: Lopullinen verkkokaupan käyttöliittymässä näkyvä HTML-rakenne esimerkkiparametreilla

Ohjelmassa 4 nähdään riveillä 1-3 div-lohko, jonka sisään lopullinen pilvipalvelun HTML-rakenne liitetään. Tämän liitoksen suorittaa app.js-pilvisovellus, joka on tuotu verkkokaupan käyttöliittymään require.js-kirjastoa hyödyntäen.

4.3 Pilvipalvelu-widgetin käytön toteutus

Tässä kappaleessa toteutetaan kaksi erilaista tapaa käyttää pilvipalvelu-widgetiä IBM WebSphere Commercessa. Näitä tapoja ovat pilvipalvelu-widgetin käyttö joko hallintatyökalun kautta tai pilvipalvelu-widgetin käyttö eSpot-toiminnallisuutta hyödyntäen.

Ensimmäinen tapa käyttää pilvipalvelu-widgetiä toteutetaan kappaleessa 4.3.1, jossa luodaan widgetille hallintatyökalutuki. Hallintatyökalutuella tarkoitetaan tässä työssä mahdollisuutta lisätä pilvipalvelu-widget verkkokaupan käyttöliittymän ulkoasusuunnitelmaan Commerce Composer -työkalun kautta. Saman työkalun kautta voidaan kertoa pilvipalvelu-widgetille, mistä pilvipalvelu haetaan.

Toinen tapa käyttää pilvipalvelu-widgetiä toteutetaan kappaleessa 4.3.2. Kyseisen kappaleen toteutuksessa pilvipalvelu-widget asetetaan manuaalisesti suoraan verkkokaupan käyttöliittymän JSP-sovelluskoodiin, jolloin pilvipalvelu-widget hakee tarvitsemansa parametrit eSpot-toiminnallisuudella.

4.3.1 Hallintatyökalutuen toteuttaminen pilvipalvelu-widgetille

Mikäli widgetin halutaan tukevan WCS:n hallintatyökalua, on sille luotava hallintatyökalun vaatimat määrittely- ja ulkoasutiedostot. Lopuksi widgetin asetukset ja määrittelyt ladataan WCS:n tietokantaan, jolloin widget on verkkokaupan käyttöliittymän ja hallintatyökalujen saatavilla. [22]

Tässä työssä hallintatyökalutuki tarkoittaa sitä, että toteutetaan hallintatyökaluun käyttöliittymäkomponentti, jonka kautta voidaan lisätä pilvipalvelu-widget verkkokaupan käyttöliittymään. Hallintatyökalu-termiä käytetään tässä työssä Management Centerin Com-

merce Composer -ulkoasun suunnittelutyökalusta. Luodun hallintatyökalun käyttöliittymän kautta voidaan myös antaa pilvipalvelu-widgetille sen tarvitsemat parametrit, jotka esiteltiin aiemmin kappaleessa 4.2.

Pilvipalvelu-widgetin käyttöliittymän tarjoaja tarvitsee siis kaksi parametria pilvipalvelun liittämiseksi käyttöliittymään: pilvipalvelun hakemiseen tarvittavat JavaScript-parametrit ja pilvipalvelun HTML-lohkon tunnistamiseen tarvittavan parametrin. Kyseiset parametrit on pystyttävä antamaan hallintatyökalun kautta pilvipalvelu-widgetin datan tarjoajalle, joka taas välittää parametrit käyttöliittymän tarjoajalle. Tästä syystä pilvipalvelu-widgetille täytyy totetuttaa tuki myös hallintatyökalulle.

Hallintatyökalutuen toteutus aloitetaan luomalla vaadittavat hallintatyökalun määrittely- ja ulkoasutiedostot. Nämä tiedostot määrittelevät hallintatyökalun käyttöliittymän, jolloin tarvittavat parametrit voidaan asettaa suoraan hallintatyökalusta. Ensimmäiseksi luodaan ohjelmassa 5 näkyvä XML-muotoinen määrittelytiedosto, jonka avulla widget saadaan käytettäväksi hallintatyökalun kautta.

```
<?xml version="1.0" encoding="UTF-8"?>

<Definitions>
   <WidgetObjectDefinition package="plm"
      definitionName="plmLayoutWidget_CloudServiceWidget"
      parentDefinitionName="plmBasePageLayoutPrimaryObjectDefinition"
      baseDefinitionName="plmBaseLayoutWidget"
      objectType="CloudServiceWidget"
      gridPropertiesDefinitionName="plmWidgetProperties_CloudServiceWidget"
      widgetDisplayGroups="AnyPage"
      iconPath="/images/pageLayouts/widgetIcons/CloudWidget.png">

      <CreateService baseDefinitionName="plmBaseCreateLayoutWidget" />
      <UpdateService baseDefinitionName="plmBaseUpdateLayoutWidget" />

      <PropertyDefinition propertyName="xWidgetProp_jsParams"
                          required="true" displayName="">
       </PropertyDefinition>

      <PropertyDefinition propertyName="xWidgetProp_containerParams"
                          required="true" displayName="">
       </PropertyDefinition>

      <Xml name="template">
         <sequence>0</sequence>
         <xWidgetProp_jsParams></xWidgetProp_jsParams>
         <xWidgetProp_containerParams></xWidgetProp_containerParams>
      </Xml>
   </WidgetObjectDefinition>
</Definitions>
```

Ohjelma 5: Hallintatyökalun määrittelytiedosto WidgetObjectDefinition.def

Ohjelma 5 koostuu parametreista, joiden avulla hallintatyökalun ohjelmistokehys saa määrittelynsä ja näiden parametrien perusteella hallintatyökalu tietää millaisia parametreja sen tulee osata välittää widgetillä. Seuraavaksi käsitellään ohjelman 5 tämän työn kannalta olennaiset kohdat.

Rivillä 10 asetetaan widgetille hallintatyökalun ryhmät. Ryhmä voi olla esimerkiksi ”SearchPage”, jolloin hallintatyökalun suodattimien avulla voidaan tarkastella vain verkkokaupan käyttöliittymän hakuosioon sopivia widgetejä. Tämän työn widgetille ryhmäksi asetetaan ”AnyPage”, jolloin pilvipalvelu-widget on haettavissa hallintatyökalun haun mistä tahansa ryhmästä. Riveillä 13 ja 14 määritellään luo- ja päivitä-palvelut widgetille, jotta widget olisi loppukäyttäjän käytettävissä hallintatyökalusta. [22]

Rivit 16-22 liittyvät parametreihin, jotka annetaan pilvipalvelu-widgetille hallintatyökalun kautta. Pilvipalvelun liittämiseen tarvitaan kaksi parametria: jsParams ja containerParams, jotka ovat määrittelytiedoston nimeämiskäytännön mukaisesti nimetty riveillä 16 ja 20. Molemmille parametrien PropertyDefinition-määrittelyille on asetettu parametri required=true, jolloin hallintatyökaluun syntyvät tekstikentät ovat pakollisia. Viimeiseksi riveillä 26-27 asetetaan jsParams ja containerParams -parametreille oletusarvot: tässä tapauksessa molemmat widgetin tarvitsemat muuttujat voidaan jättää oletusarvoiltaan tyhjiksi, sillä pilvipalvelu-widgetin tuleva toteutus osaa käsitellä virhetilanteen, jossa parametrit ovat jostain syystä jääneet asettamatta. [22] Rivin 25 ”sequence”-muuttuja määrittää widgetien näyttöjärjestyksen verkkokaupan käyttöliittymässä, jos samaan ulkoasun ennalta määritettyyn paikkaan halutaan listata useampia widgetejä. Muuttuja on asetettu oletuksena nollaan, jolloin pilvipalvelu-widget näytetään käyttöliittymässä ensimmäisenä. [23]

Ohjelmassa 6 esitellään XML-muotoinen hallintatyökalun ulkoasutiedosto, jonka perusteella hallintatyökalu osaa piirtää ulkoasuelementit parametrien ja muiden mahdollisten konfiguraatioiden antamiseen. Nämä parametrit ja konfiguraatiot ovat lopulta käytössä valmiin widgetin datan tarjoajalla, joka tarvittaessa voi välittää ne käyttöliittymän tarjoajalle. [22] Tämä ulkoasutiedosto liittyy rivin 5 ”definitionName”:n perusteella ohjelman 5 määrittelyyn.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <Definitions>
4     <GridObjectProperties
5        definitionName="plmWidgetProperties_CloudServiceWidget">
6        <PropertyPane>
7           <PropertyGroup
8                 name="widgetProperties"
9                 collapsable="false"
10                groupTitle="${plmPageLayoutResources.widgetPropertiesPrompt}">
11             <PropertyInputText
12                name="${plmPageLayoutResources.widgetNamePrompt}"
```

```
13                  propertyName="widgetName"
14                  promptText="${plmPageLayoutResources.widgetNamePrompt}"
15              <PropertyInputText
16                  propertyName="xWidgetProp_containerParams"
17                  promptText="Give container parameters" />
18              <PropertyInputText
19                  propertyName="xWidgetProp_jsParams"
20                  promptText="Give JavaScript parameters" />
21          </PropertyGroup>
22      </PropertyPane>
23    </GridObjectProperties>
24  </Definitions>
```

Ohjelma 6: Hallintatyökalun ulkoasumäärittely WidgetPropertiesView.def

Ohjelman 6 propertyGroup-lohko sisältää PropertyInputText-tageja, jotka määrittelevät näytettävät tekstikentät, joihin voidaan syöttää widgetin tarvitsemat jsParams ja containerParams -parametrien arvot. Kyseiset tagit määrittelevät hallintatyökalun käyttöliittymässä näytettävät tekstikentät jo aiemmin ohjelmassa 5 määritellyille parametreille. Myös widgetin nimelle on määritelty oma tekstikenttä, jonka avulla widget voidaan tunnistaa kaikkien hallintatyökalulla käyttöliittymään sijoiteltujen widgetien joukosta. [22]

Riveillä 16 ja 18 on määritelty hallintatyökalun tekstikenttien yläpuolella näkyvät aputekstit. Tässä työssä on käytetty staattisia tekstejä käännöstiedostojen sijaan työn selkeyttämiseksi. Lopullisessa tuotantototeutuksessa käännettävät tekstit tulee määritellä erillisiin Management Center -ohjelmistokehyksen käännöstiedostoihin. [22]

Jotta hallintatyökaluun luotu näkymä on käytettävissä, täytyy aiemmin toteutettu widget rekisteröidä hallintatyökaluun. Tämä onnistuu lisäämällä rekisteröintitiedostoon ohjelmassa 7 esitelty lohko.

```
1   <Item Delete="0">
2     <WidgetDefIdentifier>CloudServiceWidget</WidgetDefIdentifier>
3     <WidgetVendor>yritys</WidgetVendor>
4     <WidgetType>1</WidgetType>
5     <WidgetPath>
6       /Widgets/com.yritys.commerce.store.widgets.CloudServiceWidget/
7       CloudServiceWidget.jsp
8     </WidgetPath>
9     <WidgetDefinitionxml></WidgetDefinitionxml>
10    <WidgetState>1</WidgetState>
11    <WidgetDisplayName>Cloud Service Widget</WidgetDisplayName>
12    <WidgetUIObjectName>CloudServiceWidget</WidgetUIObjectName>
13    <WidgetStoreUniqueID>0</WidgetStoreUniqueID>
14    <WidgetDescription>
15      Widget pilvipalvelun liittämiseen verkkokaupan käyttöliittymään
16    </WidgetDescription>
17  </Item>
```

Ohjelma 7: Lohko pilvipalvelu-widgetin verkkokauppaan rekisteröintiin

Ohjelman 7 ensimmäisen rivin Delete-parametrin arvolla 0 widget otetaan käyttöön. Rivillä 2 asetetaan viittaus widgetin hallintatyökalun aiemmin tehtyyn määrittelyyn. Riviltä 5 alkava lohko määrittelee päätason JSP-tiedoston sijainnin, joka viittaa aiemmin luotuun päätason JSP-tiedostoon.

Rivillä 10 on asetettu widgetin tila aktiiviseksi arvolla 1, jolloin widget on verkkokaupan käytettävissä. Rivin 11 lohkossa asetetaan hallintatyökalussa näkyvä widgetin nimi, jonka avulla hallintatyökalun loppukäyttäjät löytävät widgetin muiden widgetien joukosta. Rivillä 13 on viittaus kaupan tunnisteeseen, jolle kyseinen widget halutaan rekisteröidä ja tässä tapauksessa arvoksi on asetettu 0, jolloin widget on käytössä kaikissa verkkokauppaan luoduissa kaupoissa. Viimeiseksi riveillä 14-16 on widgetin kuvausteksti, joka näkyy hallintatyökalun käyttöliittymässä. [24]

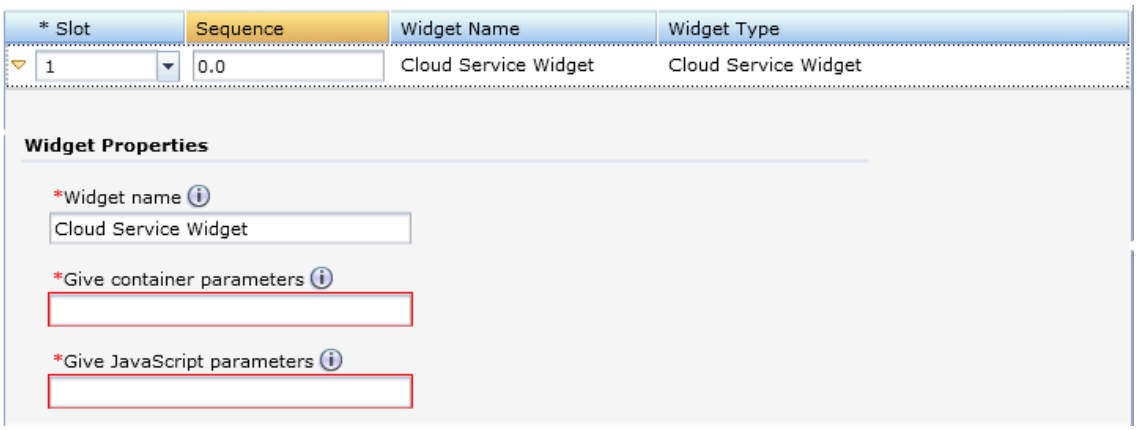
Seuraavaksi widget täytyy määritellä tilausmäärittelytiedostoon, jolloin widget on saatavilla halutulle kaupalle. Tarvittava määrittely näkyy ohjelmassa 8. Ohjelmassa 8 asetetaan rivillä 2 viite widgetin määrittelyyn, jonka tulee olla sama kuin aiemmin rekisteröintimäärittelytiedostoon asetettu viite. Kuten ohjelmassa 7, myös tässä ohjelmassa rivin 1 Delete-parametri tarkoittaa, että widget otetaan käyttöön ja rivin 3 lohko arvolla 1 kertoo widgetin olevan aktiivinen valitussa kaupassa. [24]

```
1  <Item Delete="0">
2     <WidgetDefIdentifier>CloudServiceWidget</WidgetDefIdentifier>
3     <WidgetDefinitionxml />
4     <WidgetState>1</WidgetState>
5  </Item>
```

Ohjelma 8: subscribeWidgetdef-tilausmäärittelytiedostoon lisättävä osa

Tilaus ja rekisteröinti -määrittelyjen jälkeen widget täytyy vielä ladata tietokantaan käyttämällä WCS:n tarjoamaa DataLoad-työkalua. Tätä ennen DataLoad-työkalulle on kuitenkin asetettava tarvittavat verkkokauppaprojektin ympäristömuuttujat, kuten tietokannan asetukset.

Tarvittavien ympäristömuuttujien määrittelyjen jälkeen WCS:n palvelin on pysäytettävä, jonka jälkeen dataload-työkalua käyttäen ladataan widgetin määrittelyt tietokantaan. [24] Määrittelyjen lataamisen jälkeen widget on käytettävissä hallintatyökalun kautta. Lopullinen tässä kappaleessa luotu hallintatyökalun näkymä on esitelty kuvassa 16.

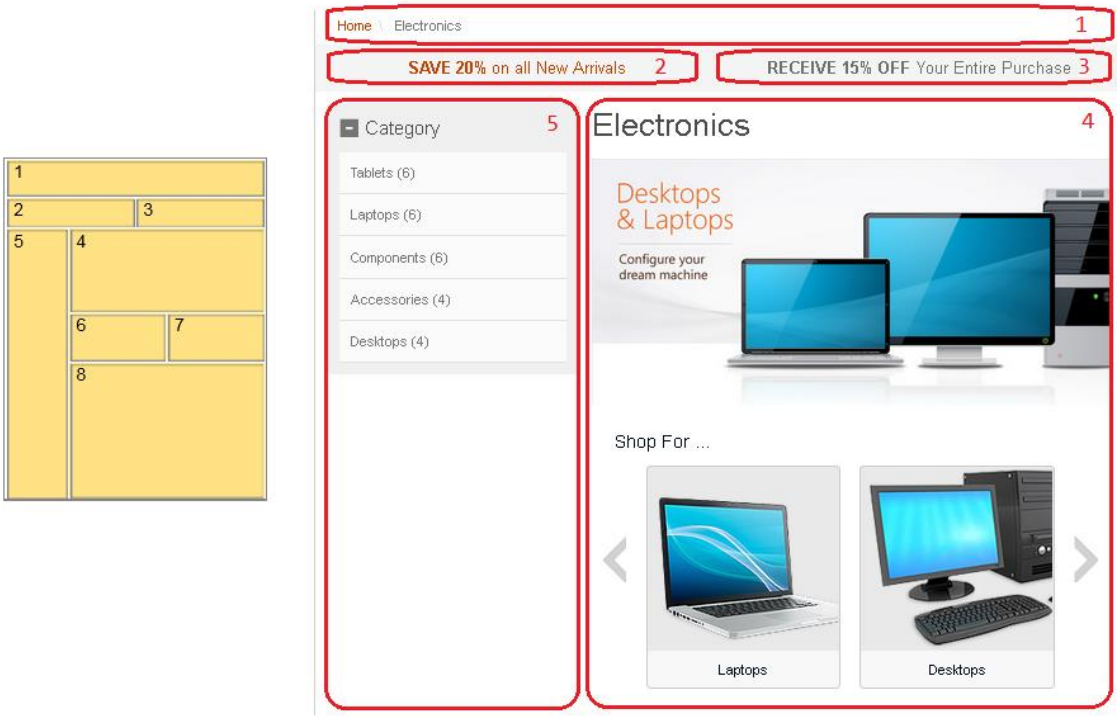


Kuva 16: Widgetin asetusten lopullinen ulkoasu hallintatyökalussa

Kuvassa 16 esiteltyyn näkymään pääsee Management Centerin Commerce Composer -työkalun kautta valitsemalla ulkoasusuunnitelman paikan, johon pilvipalvelu-widget halutaan lisätä. Tämän jälkeen hallintatyökalun hakupalkin kautta voi lisätä ”Cloud Service Widget” -nimisen pilvipalvelu-widgetin ulkoasusuunnitelmaan.

Kuvassa 16 näkyy aiemmissa määrittelytiedostoissa esiintyneet kolme tekstikenttää: widgetin nimi, containerParams ja jsParams. Pilvipalvelu-widget on asetettu ulkoasusuunnitelman paikkaan 1 ja sen sekvenssiluvuksi on asetettu 0.0, jolloin se näkyy ensimmäisenä widgetinä käyttöliittymän paikassa 1. Tallentamisen jälkeen pilvipalvelu-widget on näkyvissä verkkokaupan käyttöliittymässä.

Alkuperäisen suunnitelman mukainen pilvipalvelu-widgetin käyttäminen hallintatyökalun kautta havaittiin kuitenkin puutteelliseksi: hallintatyökalu sallii widgetin lisäämisen vain ennalta määriteltyihin paikkoihin verkkokaupan käyttöliittymässä. Kuvassa 17 esitellään yksi esimerkki mahdollisesta hallintatyökalun ulkoasusuunnitelmasta, jossa näkyvät numeroituna 1-8 paikat, joihin widget on mahdollista sijoittaa. Kuvassa 17 näkyvät myös widgetien 1-5 lopulliset paikat verkkokaupan käyttöliittymässä.



Kuva 17: Commerce Composerin ulkoasusuunnitelma ja ulkoasusuunnitelman widgetit verkkokaupan käyttöliittymässä

Kuvasta 17 havaitaan myös, että Commerce Composer rajoittaa widgetin sijoittamista verkkokaupan käyttöliittymässä ennalta määriteltyihin paikkoihin. Lisättävä sisältö pilvipalvelusta voi olla lähes mitä tahansa: pelkkä painike olemassa olevan widgetin sisällä, erillinen alasvetovalikko tai jotain muuta toiminnallisuutta yhden widgetin sisässä. Edellisten esimerkkien lisääminen Commerce Composerin kautta ei ole mahdollista, koska ulkoasusuunnitelmaan voidaan lisätä vain suurempia kokonaisuuksia verkkokaupan käyttöliittymään. Työn tavoitteena on kuitenkin luoda mahdollisimman joustava widget, joka voidaan sijoittaa mihin tahansa verkkokaupan käyttöliittymässä. Näistä syistä on tärkeää, että widgetin sijainti pystytään määrittelemään mahdollisimman tarkasti.

Joissain tapauksissa Commerce Composerin kautta lisättävä pilvipalvelu-widget voi kuitenkin olla riittävä. Esimerkiksi Commerce Composerin kautta voi lisätä loppukäyttäjälle näkyvän pilvipalveluna toteutetun etusivun mainosbannerin, joka sopii ennalta määriteltyyn paikkaan. Commerce Composerin käyttämisestä saadaan muitakin etuja, joita on käsitelty kappaleessa 4.6. Näistä syistä jo toteutettu toiminnallisuus on käyttökelpoista ja se voidaan ottaa lopulliseen widgetiin mukaan.

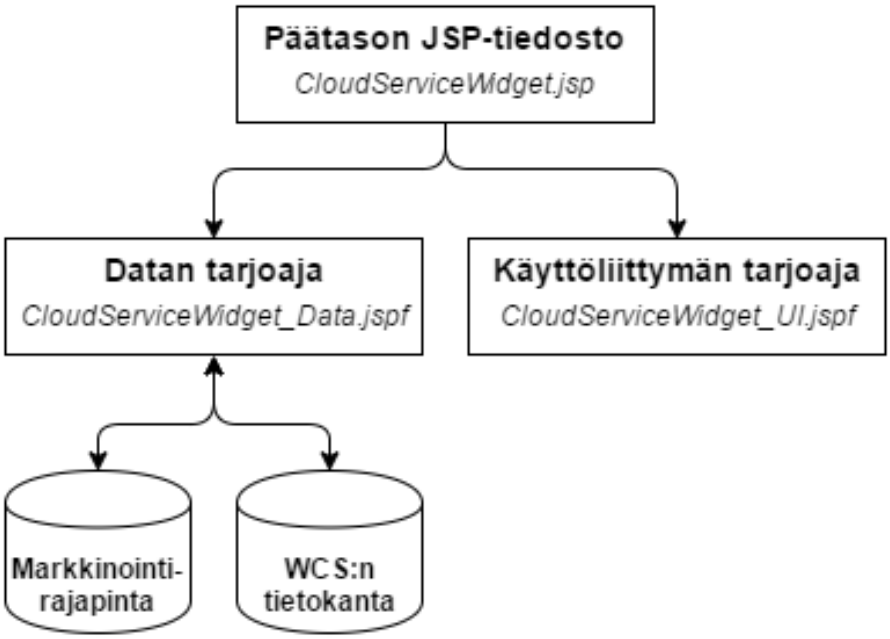
4.3.2 ESpot-tuen toteuttaminen pilvipalvelu-widgetille

Tässä kappaleessa toteutetaan vaihtoehtoinen tapa lisätä pilvipalvelu-widget verkkokaupan käyttöliittymään. Käytännössä vaihtoehtoisessa tavassa lisätään pilvipalvelu-widge-

tin JSP-tiedosto manuaalisesti haluttuun kohtaan verkkokaupan käyttöliittymässä ja annetaan pilvipalvelu-widgetin parametrit eSpot-toiminnallisuutta käyttäen. Tämän kappaleen toteutusta ei siis aseteta verkkokaupan käyttöliittymään hallintatyökalun kautta: ainoastaan parametrit annetaan hallintatyökalun eSpot-toiminnallisuutta hyödyntäen.

Edellisessä kappaleessa havaittiin jo toteutetun pilvipalvelu-widgetin toiminnallisuus puutteelliseksi, koska pilvipalvelu-widgetiä ei voida sijoittaa käyttöliittymään tarpeeksi tarkasti. Jotta pilvipalvelu-widget voitaisiin sijoittaa mihin tahansa kohtaan verkkokaupan käyttöliittymää, täytyy pilvipalvelu-widgetin parametrit hakea markkinointirajapintaa käyttäen. Käytännössä markkinointirajapinta hakee datansa eSpot-markkinointisisällöistä, joita voidaan määritellä hallintatyökalun kautta. Tätä toiminnallisuutta hyödyntäen pilvipalvelu-widgetin parametrit ovat muutettavissa käynnistämättä WCS:n palvelinta uudelleen, vaikka pilvipalvelu-widget on asetettu manuaalisesti JSP-sovelluskoodiin.

Pilvipalvelu-widgetin arkkitehtuurin mukaisesti datan tarjoajan tulee kommunikoida WCS:n markkinointirajapinnan kanssa, jotta tarvittava eSpot-toiminnallisuus saadaan toteutettua. Kuvassa 18 esitellään tarvittava lisäys aiempaan kuvan 14 pilvipalvelu-widgetin arkkitehtuuriin.



Kuva 18: Pilvipalvelu-widgetin arkkitehtuuri, jossa widget käyttää myös markkinointirajapintaa

Koko verkkokaupan käyttöliittymä muodostuu siis eri JSP-tiedostoista. Ohjelmassa 9 on esitelty sovelluskoodi, jolla eSpot voidaan lisätä mihin tahansa kohtaan JSP-tiedostoja. Ohjelman 9 rivin 3 mukaan eSpotin widgetinä käytetään pilvipalvelu-widgetiä, jolloin ohjelman 9 sovelluskoodi lisää pilvipalvelu-widgetin verkkokaupan käyttöliittymään. Koska tämä sovelluskoodi täytyy lisätä manuaalisesti JSP-tiedostoon, pilvipalvelu-wid-

getin sijoittaminen toiseen paikkaan ei onnistu hallintatyökalun kautta. Manuaalisen lisäämisen avulla pilvipalvelu on kuitenkin sijoiteltavissa mihin tahansa kohtaan verkkokaupan käyttöliittymää: esimerkiksi jo olemassa olevien widgetien sisään, jonne ulkoasusuunnitelman takia pilvipalvelua ei olisi mahdollista lisätä hallintatyökalusta.

```
<%out.flush();%>
<c:import
       url="/Widgets/com.yritys.commerce.store.widgets/CloudServiceWidget.jsp">
  <c:param name="emsName" value="CloudServiceWidget_Parameters" />
</c:import>
<%out.flush();%>
```

Ohjelma 9: eSpotin käyttäminen JSP-tiedostoissa

Käytännössä ohjelma 9 sisällyttää pilvipalvelu-widgetin halutun JSP-tiedoston joukkoon ja antaa pilvipalvelu-widgetille emsName-nimisen (e-marketing spot name) parametrin arvolla ”CloudServiceWidget_Parameters”. Kyseiselle parametrille voidaan antaa arvo hallintatyökalun kautta sen jälkeen, kun tarvittava eSpot-toiminnallisuus on toteutettu pilvipalvelu-widgetille.

Seuraavaksi toteutetaan eSpot-toiminnallisuus uuden arkkitehtuurin mukaisesti datan tarjoajalle. Koska aiemmin 4.3.1 kappaleessa toteutettu toiminnallisuus halutaan säilyttää, jätetään kyseinen toteutus uuden datan tarjoajan alkuun. Aiemman toteutuksen ehdot tarkastelevat, ovatko hallintatyökalun kautta asetetut parametrit saatavilla ja jos eivät ole, otetaan käyttöön uusi eSpot-toiminnallisuus. Kappaleessa 4.3.1 esitellyn ohjelman 2 datan tarjoajan toteutuksen perään siis lisätään ehto, joka tarkastelee, ovatko jsParams ja containerParams tyhjiä ja jos ovat, siirrytään käyttämään eSpot-toteutusta. Varsinainen eSpot-toiminnallisuuden toteutus alkaa ohjelmasta 10.

```
<%
   /* Hae eSpotin nimi pyyntö-muuttujasta ja dekoodaa se */
   String emsName = request.getParameter("emsName");
   if (emsName != null) {
      emsName = java.net.URLDecoder.decode(emsName, "UTF-8");
      request.setAttribute("emsName", emsName);
   }
%>

<c:set var="emsName" value="${requestScope.emsName}"/>
```

Ohjelma 10: eSpotin nimen hakeminen ja dekoodaaminen

Aluksi pilvipalvelu-widgetille annettu emsName-parametri on haettava ja dekoodattava request-nimisestä muuttujasta. Tämä toiminnallisuus on esitetty ohjelmassa 10. Lopuksi request-muuttujasta haettu parametrin arvo asetetaan emsName-nimiseen muuttujaan. Tässä vaiheessa emsName-muuttujan arvona on siis ”CloudServiceWidget_Parameters”, koska kyseinen emsName-muuttujan arvo välitettiin aiemmin parametrina pilvipalvelu-widgetille ohjelmassa 9.

Seuraavaksi otetaan käyttöön WCS:n markkinointirajapinta, jonka avulla haetaan hallintatyökalun eSpotiin asetetut parametrit. Koska valmiita ulkoasusuunnitelmia ei voida käyttää, täytyy parametrit asettaa eSpot-sisältöinä hallintatyökalusta, joka käsitellään myöhemmin tässä kappaleessa. ESpot-sisällön hakeminen JSP-tasolle on toteutettu ohjelmassa 11.

```
1  <%-- Hae eSpotin data palvelukutsulla --%>
2  <wcf:getData
3      type="com.ibm.commerce.marketing.facade.datatypes.MarketingSpotDataType"
4      var="marketingSpotDatas"
5      expressionBuilder="findByMarketingSpotName" >
6     <%-- eSpotin nimi --%>
7     <wcf:param name="DM_EmsName" value="${emsName}" />
8  </wcf:getData>
9
10 <wcf:eMarketingSpotCache marketingSpotData="${marketingSpotDatas}"
11                          contentDependencyName="contentId" />
12
13 <c:set var="eSpotParameters" value="" />
14
15 <c:forEach var="marketingSpotData"
16            items="${marketingSpotDatas.baseMarketingSpotActivityData}">
17    <c:set var="marketingContent"
18           value="${marketingSpotData.marketingContent.
19                    marketingContentDescription[0].marketingText}" />
20
21    <c:if test="${!empty marketingContent}">
22       <c:set var="eSpotParameters" value="${marketingContent}" />
23    </c:if>
24 </c:forEach>
```

Ohjelma 11: eSpotin datan hakeminen palvelukutsulla ja tarvittavan datan asettaminen eSpotParameters-muuttujaan

Ohjelmassa 11 käytetään WCS:n tarjoamaa palvelukutsua, joka hakee dataa WCS:n rajapinnoista. Tässä tapauksessa data haetaan markkinointirajapinnasta [25]. Palvelukutsu hakee ”CloudServiceWidget_Parameters”-nimisen eSpotin sisällön muuttujaan, johon tässä tapauksessa haetaan pilvipalvelu-widgetin tarvitsemat parametrit. Lopullinen eSpotin sisältö eli asetetut parametrit asetetaan eSpotParameters-nimiseen muuttujaan rivillä 22.

ESpotien tekstieditori on tarkoitettu vain markkinointitekstien lisäämiseen, mistä johtuen tekstieditori saattaa lisätä HTML-tageja tai eskapoida joitakin merkkejä. Siksi ohjelma 12 tarkistaa, alkaako eSpotParameters-muuttuja HTML-tagilla, josta voidaan päätellä, että muuttujan sisältö täytyy puhdistaa. Käytännössä ohjelma 12 poistaa eSpotin kautta tulleesta tekstisisällöstä tekstieditorin lisäämät ylimääräiset HTML-tagit tai merkit kuten lainausmerkit tai ”<p>”-tagit.

```
<%-- Eskapoi tagit ja merkit --%>
<c:if test="${fn:startsWith(eSpotParameters, '<p dir=\"ltr\">') }">
   <c:set var="eSpotParametersWithoutTags"
          value="${fn:substringAfter(eSpotParameters, '<p dir=\"ltr\">')}" />
   <c:set var="eSpotParametersWithoutTags"
          value="${fn:substringBefore(eSpotParametersWithoutTags, '</p>')}" />
   <c:set var="eSpotParametersWithoutTags"
          value="${fn:replace(eSpotParametersWithoutTags, '&quot;', '\"')}" />
   <c:set var="eSpotParameters" value="${eSpotParametersWithoutTags}" />
</c:if>
```

Ohjelma 12: Datan puhdistaminen ylimääräisistä HTML-tageista ja merkeistä

ESpoteihin asetettava sisältö joudutaan hallintatyökalun rajoitusten takia asettamaan yhdessä tekstikentässä. Tässä työssä tarvitaan kuitenkin kaksi parametria: containerParams ja jsParams, joten kyseiset parametrit täytyy saada välitettyä yhdessä tekstikentässä. Tässä työssä ongelma voidaan ratkaista erottelemalla parametrit putkimerkillä, jolloin tekstikenttään asetettava merkkijono on lopulta muodossa:

jsParams=data-main=https://esimerkki-url.com/app.js src=”https://esimerkki-url.com/lib/require.min.js”|containerParams=id=”cloud-service-container”

Kyseinen merkkijono täytyy parsia siten, että jsParams- ja containerParams-muuttujiin asetettava sisältö on lopulta oikeanlainen. Parametrien parsiminen eSpotille annetusta tekstikentästä on esitelty ohjelmassa 13. Käytännössä ohjelma 13 käyttää merkkijonon käsittelyyn tarkoitettuja fn-funktioita ja lopulta saa putkitetun muodon jaoteltua pilvipalvelu-widgetin tarvitsemiin jsParams- ja containerParams -muuttujiin.

```
<c:set var="splittedParameters" value="${fn:split(eSpotParameters, '|')}" />
<c:forEach var="parameter" items="${splittedParameters}">
   <c:if test="${fn:startsWith(parameter, 'jsParams')}">
      <c:set var="jsParams"
             value="${fn:substringAfter(parameter, 'jsParams=')}" />
   </c:if>
   <c:if test="${fn:startsWith(parameter, 'containerParams')}">
      <c:set var="containerParams"
             value="${fn:substringAfter(parameter, 'containerParams=')}" />
   </c:if>
</c:forEach>
```

Ohjelma 13: Parametrien asettaminen putkitetusta tekstikentästä jsParams- ja containerParams -muuttujiin

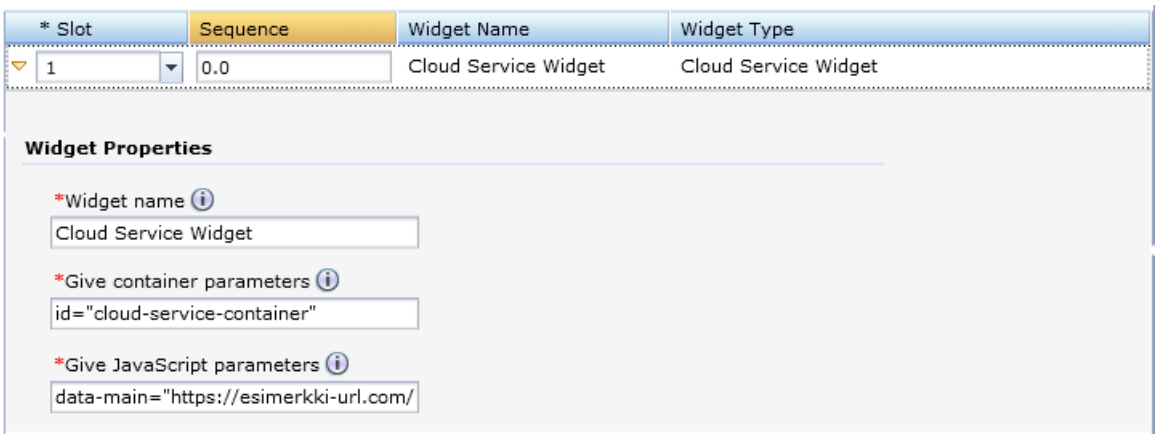
Lopullinen datan tarjoaja muodostuu yhdistämällä edellä esitetyt ohjelmat 9-13. Tämän jälkeen valmis datan tarjoaja osaa hakea markkinointirajapinnasta putkitetussa muodossa olevat parametrit sekä asettaa ne käyttöliittymän tarjoajan tarvitsemiin jsParams ja containerParams -muuttujiin oikeassa muodossa. Lopulta datan tarjoaja välittää parametrit samassa muodossa käyttöliittymän tarjoajalle kuten aiempikin hallintatyökalun kautta käytettävä toteutus. Kokonaisuudessaan toteutettu datan tarjoaja hallintatyökalutuella sekä eSpot-toiminnallisuudella esitellään liitteessä A.

4.4 Pilvipalvelu-widgetin käyttö

Tässä kappaleessa esitellään toteutetun pilvipalvelu-widgetin käyttöä kahdella aiemmin toteutetulla tavalla. Kappaleessa 4.4.1 esitellään pilvipalvelu-widgetin käyttöä hallintatyökalun kautta ja kappaleessa 4.3.2 sen sijaan keskitytään pilvipalvelu-widgetin käyttöön eSpot-toiminnallisuutta hyödyntäen.

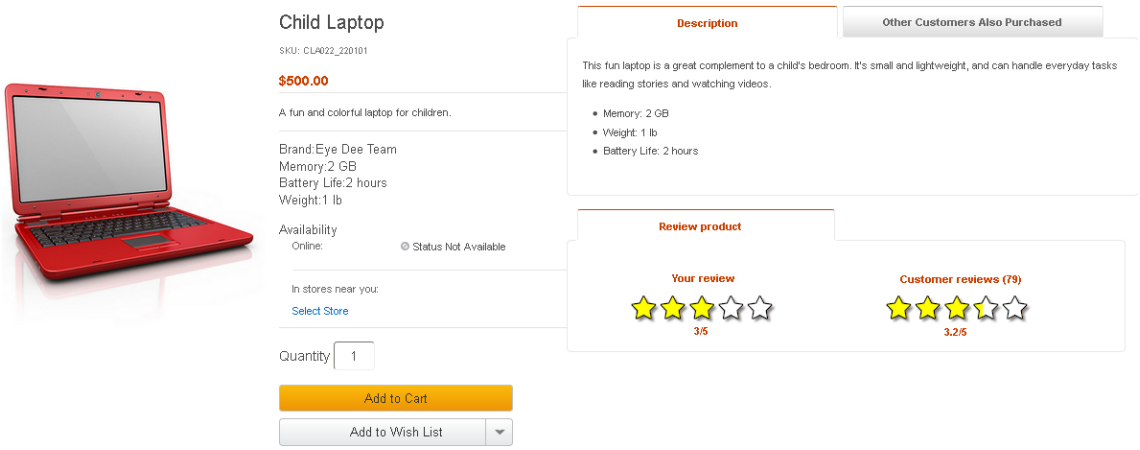
4.4.1 Pilvipalvelu-widgetin käyttö hallintatyökalun kautta

Pilvipalvelu-widgetillä voidaan kappaleessa 4.3.1 toteutetun tuen avulla lisätä pilvipalveluita käyttöliittymään WCS:n Management Center -hallintatyökalun kautta. Käytännössä tämä tapahtuu Management Centerin Commerce Composer -välilehden kautta, josta valitaan haluttu ulkoasusuunnitelma ja navigoidaan ulkoasusuunnitelman design layout -välilehdelle. Ulkoasusuunnitelman ennalta määritetyistä paikoista valitaan haluttu paikka ja lisätään hakua käyttäen Cloud Service Widget eli pilvipalvelu-widget siihen. Tässä vaiheessa näkymänä on pilvipalvelu-widgetin hallintatyökaluun aiemmin kappaleessa 4.3.1 toteutettu näkymä, joka on esitelty kuvassa 19.



Kuva 19: Pilvipalvelu-widgetin hallintatyökaluun toteutettu näkymä

Kuvassa 19 nähdään aiemmin toteutetut kolme tekstikenttää, joihin on asetettu tarvittavat parametrit, jotta esimerkkisovellus saadaan lisättyä verkkokaupan käyttöliittymään. Näkymän tallentamisen jälkeen toteutettu pilvipalvelu on näkyvissä verkkokaupan käyttöliittymässä esimerkiksi kuvan 20 tavalla.



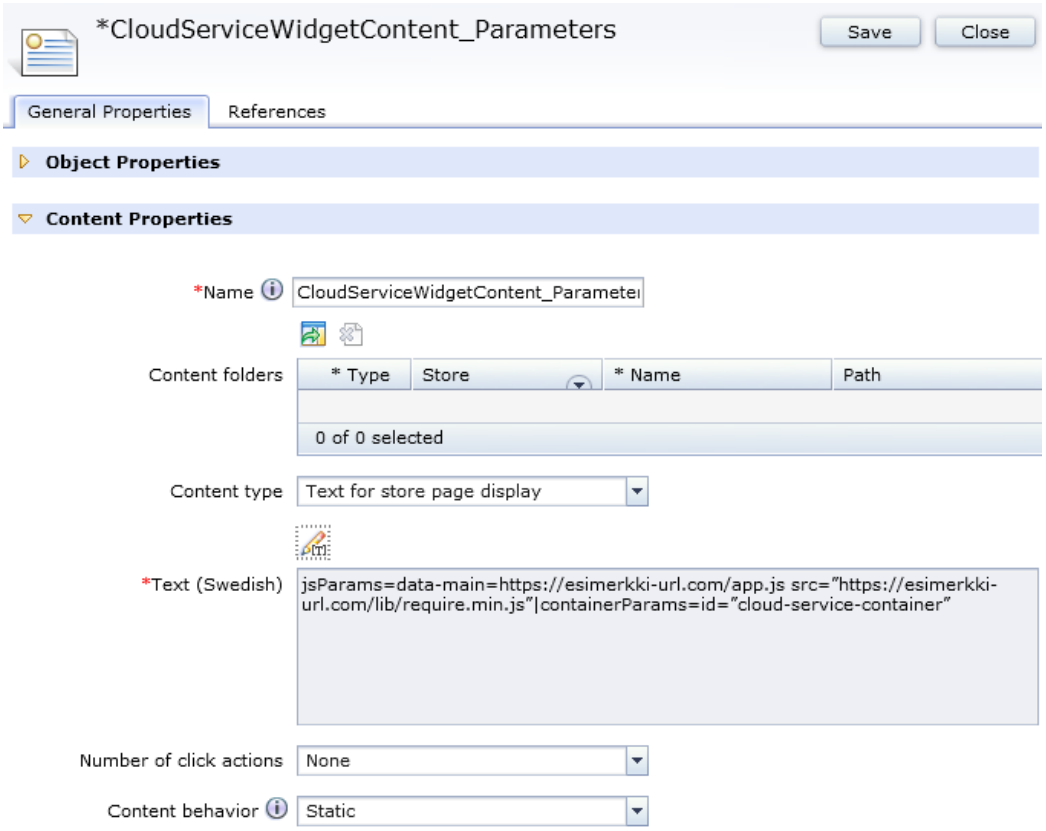
Kuva 20: Lopullinen verkkokaupan tuotesivun käyttöliittymä, johon on lisätty pilvipalvelu-widget Commerce Composer -hallintatyökalun kautta

Kuvassa 20 pilvipalvelu-widget on asetettu ulkoasusuunnitelman valmiiseen paikkaan (”Review product” -widget oikeassa alareunassa). Pilvipalvelu-widget tuo käyttöliittymään sovelluksen, jolla on mahdollista arvioida tuote ja nähdä muiden asiakkaiden tekemät arviot. Kuvan muut widgetit ovat tavallisia IBM WebSphere Commercen widgetejä.

4.4.2 Pilvipalvelu-widgetin käyttö eSpot-toiminnallisuutta hyödyntäen

Tässä kappaleessa esitellään, kuinka pilvipalvelu-widgetiä käytetään eSpot-toiminnallisuudella, joka ei rajoita pilvipalvelu-widgetin sijoituspaikkaa verkkokaupan käyttöliittymässä. Jotta pilvipalvelu-widgetiä voidaan käyttää eSpotina, täytyy eSpot ensin lisätä haluttuun paikkaan verkkokaupan käyttöliittymässä aiemmin kappaleessa 3.2.1 esitellyn ohjelman 9 tapaan.

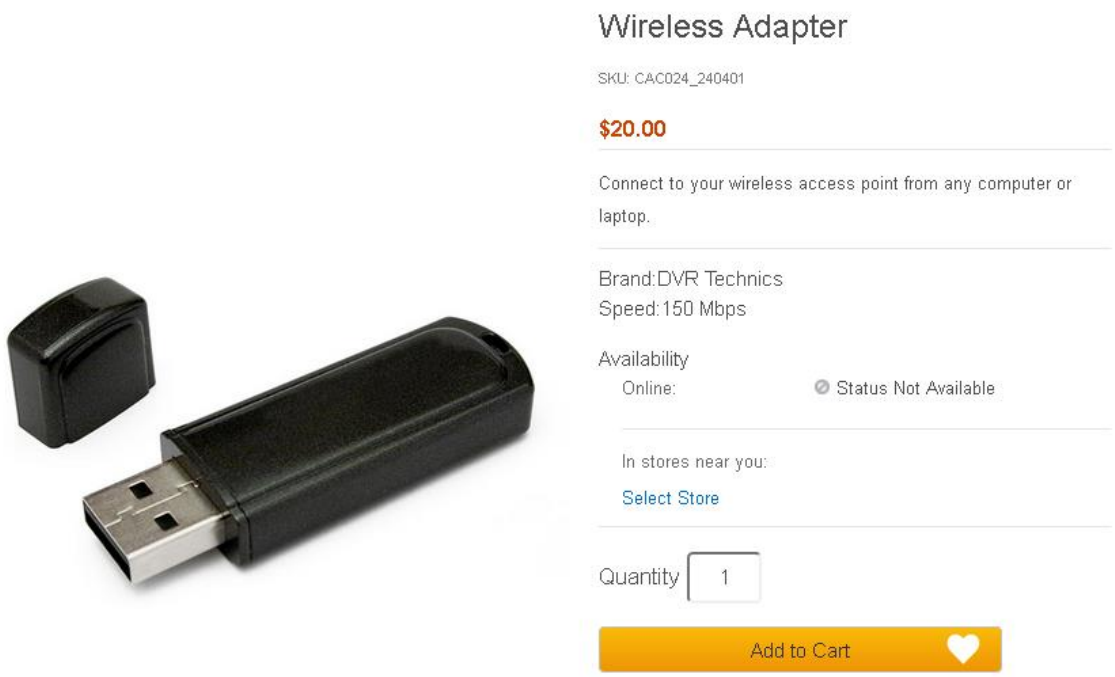
Manuaalisen sijoittamisen jälkeen välitettävät parametrit täytyy antaa eSpot-sisältönä hallintatyökalusta. Tämä onnistuu siirtymällä hallintatyökalun markkinointivälilehdelle, jossa luodaan uusi sisältösivu. Sisällön tyypiksi asetetaan tavallinen tekstikenttä, jotta parametrit voidaan antaa aiemmin esitellyssä putkitetussa muodossa. Tarvittavat parametrit asetettuna markkinointisisällön luomisnäkymään esitellään kuvassa 21.



Kuva 21: Näkymä, jossa luodaan markkinointisisältöjä ja johon on asetettu tarvittavat parametrit pilvipalvelun liittämiselle

Tämän jälkeen on luotava vielä eSpot, johon aiemmin luotu markkinointisisältö liitetään. ESpot luodaan samasta hallintatyökalun markkinointinäkymästä ja sen nimeksi tulee asettaa ”CloudServiceWidget_Parameters”, kuten aiemmin ohjelmassa 9 annettiin datan tarjoajalle välitettäväksi eSpotin nimeksi. Tällä tavoin tekstikenttään asetetut parametrit saadaan välitettyä pilvipalvelu-widgetille.

Lopulta luodun eSpotin sisällöksi asetetaan äskettäin luotu kuvassa 21 esiintyvä markkinointisisältö. Tällöin ohjelmassa 9 lisätty pilvipalvelu-widget osaa hakea oikean markkinointisisällön eli tässä tapauksessa putkitetun parametrit sisältävän tekstikentän sisällön. Kuvassa 22 näkyy pilvipalvelu liitettynä verkkokaupan tuotesivulle eSpot-toiminnallisuutta käyttäen.



Kuva 22: Lisää toivelistaan -painike (sydän-ikoni lisää ostoskoriin -painikkeessa) pilvipalveluna verkkokaupan käyttöliittymässä eSpot-toiminnallisuudella liitettynä

Pilvipalvelu on kuvan tapauksessa käyttöliittymäkomponentin sisään asetettu sydän-painike, jollaisen lisääminen hallintatyökalun ulkoasusuunnitelmaan ei ole mahdollista. Tässä tapauksessa lisää ostokoskoriin -painikkeeseen upotetulla sydän-painikkeella voi lisätä tuotteen toivelistaan.

4.5 IBM WebSphere Commerceen liitettävä pilvipalvelu

Tässä kappaleessa käsitellään, mitä rajoituksia pilvipalvelun toteutuksella on pilvipalvelu-widgetiä käytettäessä ja kuinka pilvipalvelu voi kommunikoida IBM WebSphere Commercen kanssa rajapintojen kautta.

Luvussa 2 havaittiin työn vaatimuksiin nähden sopivimmaksi arkkitehtuuriksi mikropalveluarkkitehtuuri. Pilvipalvelu toteutetaan noudattaen mikropalveluarkkitehtuurin periaatteita, jolloin palvelut ovat helposti ylläpidettävissä ja julkaistavissa. Mikropalveluarkkitehtuurin käyttämisen myötä myös kehitystyökalut ja -tavat voidaan valita vapaammin.

4.5.1 Rajoitteet pilvipalvelun toteutuksessa

Käytännössä pilvipalvelu voidaan toteuttaa mille tahansa pilvipalveluntarjoajan alustalle ja millaisilla teknologioilla tahansa. Pilvipalvelu-widgetin käytöstä seuraa kuitenkin joitakin rajoituksia. Näitä rajoituksia esitellään tässä kappaleessa.

Pilvipalvelu-widgetin on haettava pilvipalvelussa tarjotut HTML-tiedostot verkkokaupan käyttöliittymään. Tämä aiheuttaa pilvisovelluksen toteutukselle rajoituksen: pilvisovelluksen tulee tuoda HTML-rakenne verkkokaupan käyttöliittymään. Tätä varten pilvipalvelu-widgetin käyttöliittymän tarjoajassa on div-lohko, joka näkyy aiemmin kappaleessa 4.2 esitellyssä ohjelmassa 4. Tähän div-lohkoon täytyy tuoda lopullisen pilvipalvelun käyttöliittymä, joten pilvisovelluksen käyttöliittymäkoodissa siis täytyy tuoda HTML-rakenne lohkon sisään. Tämä toiminnallisuus on toteutettavissa esimerkiksi jQuery-JavaScript-kirjastolla kuten ohjelmassa 13 on esitelty.

```
1  var $container = $('#cloud-service-container');
2  $container.html(htmlTemplateExample);
```

Ohjelma 13: Pilvipalvelun käyttöliittymän sisällyttäminen cloud-service-container-lohkoon

Rivillä 1 haetaan $container-muuttujaan pilvipalvelu-widgetille parametrina annettu cloud-service-container-id:llä tunnistettavissa oleva div-lohko. Rivillä 2 sisällytetään div-lohkoon pilvipalvelun HTML-rakenne, jolloin käyttöliittymä on nähtävissä loppukäyttäjille.

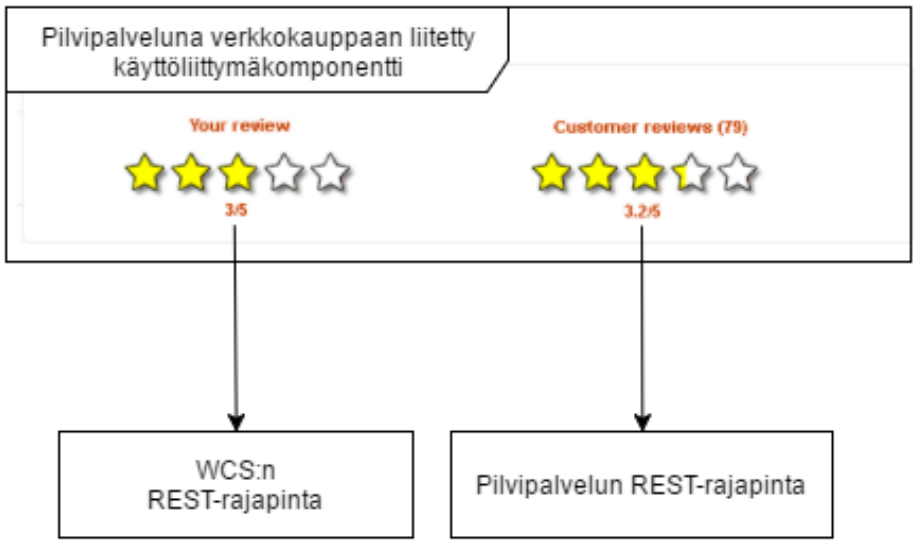
Jos pilvipalvelulle toteutetaan oma palvelin, tulee pilvipalvelun käyttämälle rajapinnalle sallia Cross Origin Resource Sharing (CORS) [26]. CORS on tapa sivuuttaa saman alkuperän käytäntö (engl. Same Origin Policy), joka taas on selaimen tapa rajoittaa yhden sivuston tekemiä pyyntöjä toisiin sivustoihin. Tässä kontekstissa sivusto koostuu domain-nimestä, protokollasta ja porttinumerosta. CORS on yleensä helposti sallittavissa palveluille pilvipalveluntarjoajan selainkäyttöliittymästä. CORS on sallittava, koska pilvipalvelu tuodaan verkkokaupan käyttöliittymään ja se tekee pyyntönsä verkkokaupan domain-nimestä. Tämä siis tarkoittaa, että ainakin verkkokaupan domain-nimestä tulevat pyynnöt on sallittava pilvipalvelun palvelinpäässä, jos pilvipalvelun täytyy käyttää omaa palvelintaan esimerkiksi rajapintakutsuihin.

4.5.2 Pilvipalvelun kommunikointi rajapinnoilla IBM WebSphere Commercen kanssa

IBM WebSphere Commercen versio 7 Feature Pack 8:lla tarjoaa REST-rajapintoja käytettäväksi kommunikointiin WCS:n palvelinpään kanssa [27]. REST on arkkitehtuurityyli, jonka tarkoituksena on helpottaa hajautettujen järjestelmien hallintaa ja luomista.

Verrattuna vanhempiin rajapintatoteutuksiin, kuten XML-RCP tai SOAP, REST on täysin teknologiariippumaton eikä tarvitse asiakaspäältä erityistä toiminnallisuutta. Ainoa vaatimus REST-rajapinnoille on, että asiakassovellus pystyy kommunikoimaan jollain protokollalla rajapinnan kanssa [10]. Tästä syystä IBM WebSphere Commercen REST-rajapinnat ovat helposti otettavissa käyttöön myös pilvipalvelussa.

WCS tarjoaa useita eri REST-rajapintoja liittyen esimerkiksi verkkokaupan tuotteisiin, markkinointiominaisuuksiin tai käyttäjiin. Näitä rajapintoja käyttämällä voidaan esimerkiksi hakea tuotedataa, tehdä uusia tilauksia tai hallita verkkokaupan käyttäjiä. [27] Esimerkki pilvipalvelun kommunikoinnista WCS:n ja oman rajapintansa kanssa esitellään kuvassa 23, jossa kuvan esimerkkipilvipalvelu hakee käyttäjän tekemät arviot valitusta tuotteesta WCS:n rajapinnasta, kun muiden käyttäjien arviot se taas hakee pilvipalvelun omasta REST-rajapinnasta. Tässä esimerkissä pilvipalvelulle on toteutettu REST-rajapinta, mutta tämän työn toteutus ei rajaa muunlaisiakaan rajapintatoteutuksia pois.



Kuva 23: Pilvipalvelu kommunikoi WCS:n rajapinnan ja oman REST-rajapintansa kanssa

Mikäli tarvitaan uutta toiminnallisuutta, jota WCS:n rajapinta ei suoraan tarjoa, voidaan toteuttaa tarvittava rajapinta verkkokauppaan myös itse. Tällaista toiminnallisuutta voisi olla esimerkiksi rajapinta, jonka kautta voi antaa ja hakea palautetta verkkokaupasta tai kuvassa 23 esitelty toiminnallisuus, jossa käyttäjä hakee oman arvionsa WCS:n REST-rajapinnalla. Usein verkkokauppaa kehitettäessä ja muutoksia tehdessä jo olemassa olevia rajapintoja saatetaan joutua laajentamaan tai muuttamaan, johon WCS:n arkkitehtuuri tarjoaa mahdollisuuden sovelluskehittäjille. Uusien rajapintojen luominen tai olemassa olevien muuttaminen vaatiikin muutoksia Java-sovelluskoodiin ja XML-konfiguraatioihin. [27]

Pilvipalvelu voi kommunikoida käytännössä minkä tahansa rajapinnan kanssa: oman rajapintansa, WCS:n rajapinnan tai esimerkiksi jonkin toisen mikropalvelun rajapinnan kanssa. Pilvipalvelun ei kuitenkaan ole pakko käyttää rajapintoja, vaan se voi olla pelkkä käyttöliittymäkomponentti, jota kehitetään pilvikehitysmallista saatavien hyötyjen takia irrallisena käyttöliittymäkomponenttina.

4.6 Ratkaisun arviointi

Tässä kappaleessa arvioidaan ja vertaillaan pilvipalvelu-widgetin kahta toteutettua käyttötapaa sekä pohditaan millaisissa tilanteissa eri tapoja kannattaa käyttää. Kappaleen lopussa pohditaan myös pilvipalvelun toteutusarkkitehtuurista syntyneitä hyötyjä ja niiden vaikutusta työhön.

Etuina pilvipalvelu-widgetin käyttämiselle hallintatyökalun kautta on esimerkiksi se, että pilvipalvelu voidaan lisätä verkkokaupan käyttöliittymään julkaisematta koko sovellusta. Myös yrityskäyttäjät voivat hallinnoida eri widgetejä tätä kautta, jolloin sovelluskehittäjiä ei tarvita pilvipalvelun lisäämiseen tai sen siirtämiseen verkkokaupan käyttöliittymässä. Widgetin lisääminen tai siirtäminen ei myöskään tässä tapauksessa tarvitse sovelluksen julkaisua. Lisäksi parametrien antaminen hallintatyökalun kautta on helppoa, koska hallintatyökaluun luotiin kappaleessa 4.3.1 kolmesta tekstikentästä koostuva näkymä, jonka avulla on yksinkertaista asettaa pilvipalvelu-widgetin tarvitsemat parametrit.

Pilvipalvelu-widgetin lisääminen hallintatyökalun kautta kuitenkin rajoittaa pilvipalvelun sijoittamisen vain ulkoasusuunnitelman ennalta määriteltyiihin paikkoihin. Ulkoasusuunnitelmat ovat usein jaoteltu melko suuriin kokonaisuuksiin, joten esimerkiksi yhden käyttöliittymäkomponentin sisällä olevia pienempiä kokonaisuuksia, kuten yksittäisiä painikkeita, ei voida lisätä pilvipalveluna käyttöliittymään käyttäen hallintatyökalua.

Hallintatyökalutukeen verrattuna suurin hyöty pilvipalvelu-widgetin eSpot-toiminnallisuudesta on se, että pilvipalvelu on asetettavissa mihin tahansa verkkokaupan käyttöliittymään. Tämän ansiosta pilvipalvelu-widget on joustavampi ja käytettävissä myös tilanteissa, joissa pilvipalvelu ei suoraan sovi valmiiseen ulkoasusuunnitelmaan.

Luodun eSpot-toiminnallisuuden takia menetetään kuitenkin joitain etuja verrattuna hallintatyökalun kautta lisäämiseen. Pilvipalvelun lisääminen verkkokaupan käyttöliittymään tai sen siirtäminen uuteen sijaintiin vaatii sovelluskehittäjien tekemiä muutoksia JSP-tiedostoihin ja vaatii koko sovelluksen julkaisun. Myös parametrien antaminen on hankalampaa parametrien putkitetun muodon takia. Parametreja voidaan kuitenkin vaihtaa julkaisematta tai uudelleenkäynnistämättä WCS:ää hallintatyökalun kautta. Tämä mahdollistaa esimerkiksi pilvipalvelun vaihtamisen toiseen pilvipalveluun siinä kohdassa, johon eSpot on asetettu.

Verkkokaupan käyttöliittymään pilvipalvelu-widgetillä liitettävä pilvipalvelu toteutetaan mikropalveluperiaatteita noudattaen. Tällöin pilvipalvelusta tulee itsenäisesti kehitettävä ja julkaistava palvelu, jonka seurauksena palvelun ei tarvitse noudattaa IBM WebSphere Commercen julkaisusykliä. Toteutettavat palvelut ovat myös riippumattomia IBM WebSphere Commercesta tai toisista palveluista, mikä helpottaa ja nopeuttaa kehitys-

työtä. Koska palvelut ovat teknologiariippumattomia, myös kehitysteknologiat ja -ympäristöt ovat vapaasti valittavissa. Näistä eduista johtuen pilvipalvelun kehitystyö vastaa tässä työssä asetettuja vaatimuksia.

Tässä työssä pilvipalvelun liittämiseen toteutettiin kaksi eri tapaa: lisääminen manuaalisesti mihin tahansa verkkokaupan käyttöliittymään tai lisääminen hallintatyökalun kautta ennalta määriteltyihin ulkoasusuunnitelman paikkoihin. Pilvipalvelu-widgetin käytön prosessin kannalta tämä hieman hankaloittaa päätöstä, kummalla tavalla pilvipalvelu tietyssä tilanteessa kannattaa lisätä, mutta samalla myös lisää pilvipalvelu-widgetin joustavuutta ja helppokäyttöisyyttä. Jos käyttöliittymäkomponentti sopii ulkoasusuunnitelman ennalta määriteltyyn paikkaan, voidaan lisäämiseen käyttää hallintatyökalua. Jos taas käyttöliittymäkomponentti ei sovi suoraan ulkoasusuunnitelmaan, lisätään pilvipalvelu manuaalisesti verkkokaupan käyttöliittymään ja annetaan sen parametrit markkinointirajapintaa käyttäen.

Työn toteutuksen jälkeen syntyi myös melko hyvä kuva siitä, minkälaiset palvelut kannattaa suoraan toteuttaa ulkoisena pilvipalveluna. Tällaisia olivat palvelut, jotka eivät ole vahvasti riippuvaisia verkkokaupan ydintoiminnallisuudesta, kuten toivelista tai asiakastuki-sovellus. Myös palvelut, joiden tarpeisiin on tarjolla hyvä tuki WCS:n REST-rajapinnoissa, voidaan melko helposti toteuttaa pilvipalveluina. Hyvin vahvasti erilaisiin verkkokaupan ydintoimintoihin sidotut toiminnot, kuten käyttäjienhallinta tai tilaustenhallinta, ovat haastavampia toteuttaa pilvisovelluksen kanssa toimiviksi.

5. YHTEENVETO JA JATKOKEHITYS

Tässä luvussa esitellään työn pääkohdat ja tärkeimmät tulokset. Luvussa pohditaan myös sitä, millaisilla tavoilla pilvipalvelu-widgetiä voitaisiin jatkokehittää.

5.1 Yhteenveto

Tässä työssä etsittiin ratkaisua IBM WebSphere Commerce -verkkokauppa-alustan käyttöliittymäkehityksen tehostamiseksi ja helpottamiseksi. Työssä tutkittiin myös, olisiko käyttöliittymäkehitystä mahdollista tehdä riippumatta IBM WebSphere Commercesta ja sen käyttämistä teknologioista.

Työssä esiteltiin websovellusarkkitehtuureita ja pohdittin, kuinka erilaiset arkkitehtuurit sopivat käyttöliittymätason laajentamiseen. Lopulta työssä ratkaistiin käyttöliittymäkehityksen ongelmat liittämällä käyttöliittymäkomponentteja pilvipalveluina IBM WebSphere Commerce -verkkokauppa-alustan käyttöliittymäkerrokselle. Ratkaisun avulla käyttöliittymäkehitystä on mahdollista tehdä teknologiariippumattomasti. Työssä toteutettu ratkaisu teki käyttöliittymäkehityksestä myös tehokkaampaa ja kehittäjäystävällisempää.

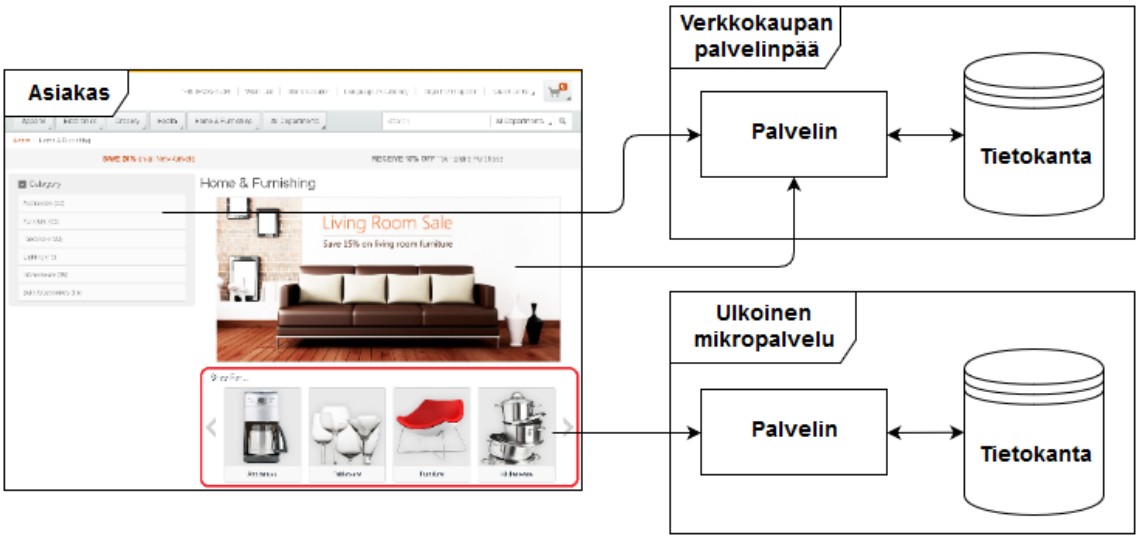
Tämän työn totetutus on osittain yleistettävissä websovelluksien käyttöliittymän laajentamiseen. Alustan ei tarvitse olla IBM WebSphere Commerce, mutta käyttöliittymän tulisi jollain tasolla muodostua yksittäisistä käyttöliittymäkomponenteista, jotta pilvipalvelu-widgetillä voitaisiin lisätä järkeviä kokonaisuuksia käyttöliittymään. Pilvipalvelu-widgetin asiakaspään toteutus rajoittuu JSP-teknologiaan, mutta toteutus on melko helposti muutettavissa asiakaspään käyttöliittymän laajentamiseen pilvipalveluilla, jolloin käytännössä minkä tahansa verkkosivun asiakaspäähän voisi liittää pilvipalvelun tämän työn ratkaisua soveltaen.

IBM WebSphere Commerce on laaja kokoelma erilaista liiketoimintalogiikkaa, josta suurin osa ei välttämättä ole käytössä tai tarpeellista toteutettavan verkkokaupan kannalta. Koska kaikki toiminnallisuus on toteutettu yhteen järjestelmään, muodostuu WCS:stä hankalasti ylläpidettävä ja kehitettävä monoliittinen järjestelmä. Järjestelmän suuri koko aiheuttaa myös julkaisuprosessiin ongelmia, sillä laajuutensa vuoksi WCS:n eri osat ovat vahvasti riippuvaisia toisistaan, jolloin julkaisuissa syntyvät ongelmat ovat huomattavasti todennäköisempiä kuin pienempien sovellusten tapauksessa. WCS:n käyttämät ohjelmistokehykset ja teknologiat ovat osittain vanhentuneita. Tämän seurauksena kehitystyö on hyvin hidasta eivätkä kehitystyökalut ole nykypäivän ketterän kehityksen vaatimuksia vastaavia. WCS on myös teknologiariippuvainen, mikä käytännössä tarkoittaa sitä, että verkkokauppa-alustaa on kehitettävä Java EE:llä ja käyttöliittymää taas JSP-teknologioilla.

Ratkaisua näihin ongelmiin etsittiin tutkimalla erilaisia websovellusarkkitehtuureita kuten kolmikerrosarkkitehtuuria, palvelupohjaista arkkitehtuuria sekä mikropalveluarkkitehtuuria. Kolmikerrosarkkitehtuuri todettiin IBM WebSphere Commercen nykyiseksi arkkitehtuuriksi, joka järjestelmän nykyisten ongelmien takia ei ole paras arkkitehtuuri laajentamiseen. Palvelupohjainen arkkitehtuuri sen sijaan koettiin melko hyväksi arkkitehtuuriksi käyttöliittymätason laajentamiseen sen teknologiariippumattomuuden ansiosta. Lopulliseksi arkkitehtuuriksi valittiin kuitenkin mikropalveluarkkitehtuuri, joka sekin on palvelupohjainen arkkitehtuuri tietyillä ominaisuuksilla, kuten vähemmillä riippuvuuksilla toisiin komponentteihin sekä pienempikokoisilla palveluilla.

Ratkaisuna lopulliseen käyttöliittymäkehityksen ongelmaan työssä esiteltiin pilvipalvelujen liittäminen IBM WebSphere Commercen käyttöliittymään. Liitettävät pilvipalvelut toteutettiin mikropalveluarkkitehtuurilla, jolloin päästiin eroon suurimmasta osasta WCS:n aiheuttamista käyttöliittymäkehityksen ongelmista.

WCS:n koko käyttöliittymä koostuu käyttöliittymäkomponenteista eli widgeteistä. WCS:ään täytyy luoda erityinen widget, joka tässä työssä on nimeltään pilvipalvelu-widget. Tällöin pilvipalvelu-widget saa tavallisen widgetin ominaisuudet kuten hallintatyökalutuen. Pilvipalvelu-widget hakee ulkoisesti toteutetun pilvipalvelun WCS:n käyttöliittymään ja suorittaa sen. Kuvassa 24 esitellään yksinkertaistettu arkkitehtuuri, jossa käyttöliittymäkomponentti on haettu WCS:n käyttöliittymään ulkoisesta mikropalvelusta, kun taas muut käyttöliittymäkomponentit haetaan normaaliin tapaan verkkokaupan palvelinpäästä.



Kuva 24: Arkkitehtuuri ulkoisena palveluna toteutetun pilvipalvelun liittämiselle WCS:n käyttöliittymään

Lopullinen ratkaisu mahdollistaa pilvipalvelujen liittämisen IBM WebSphere Commercella toteutetun verkkokaupan käyttöliittymään joko hallintatyökalusta ulkoasusuunnitelman ennalta määriteltyihin paikkoihin tai lisäämällä pilvipalvelun manuaalisesti mihin

tahansa kohtaan käyttöliittymää eSpot-toiminnallisuudella. Tämän ratkaisun avulla pilvipalveluita voidaan kehittää vapaasti valituilla teknologioilla, jolloin päästään eroon IBM WebSphere Commercen aiheuttamista teknologiariippuvuuksista. Pilvipalvelut toteutetaan mikropalveluarkkitehtuuria noudattaen, jolloin niiden ei tarvitse noudattaa IBM WebSphere Commercen kehitys- tai julkaisusykliä. Samaa pilvipalvelu-widgetiä käyttämällä voidaan lisätä myös useita eri pilvipalveluita verkkokaupan käyttöliittymään.

Lopullinen toteutettu työ vietiin myös oikean verkkokaupan tuotantoympäristöön, jossa se todettiin toimivaksi tavaksi liittää pilvipalvelu verkkokaupan käyttöliittymään. Työn pilvipalvelu-widgetin toteutus siis onnistui hyvin, vaikkakin IBM WebSphere Commerce aiheutti työn toteutukseen joitakin ongelmia ja rajoituksia, kuten välitettävien parametrien parsimisen, sekä haasteita lopullisen käyttöliittymäkomponentin sijoitteluun verkkokaupan käyttöliittymässä. Kyseiset ongelmat saatiin tämän työn toteutusaikataulun puitteissa ratkaistua, mutta ne jätettiin kuitenkin avoimiksi jatkokehitykselle.

Valmista pilvipalvelu-widgetiä käytettäessä osa IBM WebSphere Commercella toteutetun verkkokaupan käyttöliittymäkehityksestä pystyttiin erottamaan teknologiariippumattomaksi pilvipalvelukehitykseksi. Myös kehitystyöhön kulunut aika pieneni, sillä ensimmäinen oikea pilvipalvelutoteutus ja sen liittäminen WCS:ään pilvipalvelu-widgetillä kesti saman verran kuin arvioitiin saman toiminnallisuuden toteuttamisen suoraan verkkokauppa-alustaan kestävän. Pilvipalvelun liittäminen verkkokaupan käyttöliittymään pilvipalvelu-widgetillä ei myöskään aiheuttanut ongelmia muihin verkkokaupan käyttöliittymäkomponentteihin tai WCS:n muihin osiin.

Lähimpänä vertailukohtana tässä työssä toteutetulle käyttöliittymäkomponenttien irrottamiselle palveluiksi voidaan pitää erilaisia mashup-palveluita. Mashup-palvelulla tarkoitetaan palvelua, joka on useista eri palveluista koostettu kokonaisuus. Esimerkki tällaisesta käyttöliittymätasolle liitettävästä palvelusta on esimerkiksi Google Maps. Kuten tässäkin työssä toteutetut pilvisovellukset, myös Google Maps tarjoaa sekä valmiin käyttöliittymäkomponentin liitettäväksi sivulle että oman rajapinnan karttatietoihin. Koska erilaiset mashup-palvelut ja niissä tehty käyttöliittymätason integrointi on hyvin yleistä ja todettu toimivaksi ratkaisuksi, voidaan myös tämän työn toteutusta pitää onnistuneena ratkaisuna. [28]

5.2 Pilvipalvelu-widgetin jatkokehitys

Vaikka pilvipalvelu-widgetin toteutus todettiin hyväksi ratkaisuksi laajentaa verkkokaupan käyttöliittymää ja helpottaa kehitystyötä, työtä voidaan silti jatkokehittää. Suurin osa jatkokehityksestä onkin pilvipalvelu-widgetin käyttöä helpottavia ja sen joustavuutta lisääviä ominaisuuksia.

Pilvipalvelu-widgetin eSpot-toiminnallisuudessa olisi hyvä olla erillinen eSpotin content-kenttä parametreille hallintatyökalussa. Tällöin parametreja ei tarvitsisi antaa tavalliseen

tekstikenttään vaan kustomoituun content-kenttään, joka sisältäisi kaksi tekstikenttää, joihin annettaisiin muuttujien arvot. Näin myös päästäisiin eroon parametrien hankalasti muodostettavasta putkitetusta muodosta. Mikäli hallintatyökalua ei haluta kustomoida uudella content-kentällä, voitaisiin parametrit antaa esimerkiksi helpommassa XML-muodossa. Putkitettua muotoa päädyttiin käyttämään työssä, koska tässä työssä käytetty WCS:n versio ei tukenut JSP-tiedostoissa XML-muotoisen datan parsimista.

Pilvipalvelu-widget voisi tukea myös paremmin WCS:n markkinointiominaisuuksia, esimerkiksi erilaisia aktiviteetteja kuten eri sisällön näyttämistä eri käyttäjäryhmille. On mahdollista, että tämä toiminnallisuus voitaisiin toteuttaa esimerkiksi käyttämällä WCS:n seuraavien versioiden REST-rajapintoja, jotka tukevat paremmin esimerkiksi hallintatyökalujen ja eri markkinointiominaisuuksien kanssa kommunikointia. Tämä toiminnallisuus olisi todennäköisesti mahdollista myös toteuttaa pilvipalvelun päähän tunnistamalla eri asiakassegmenttejä tai muita markkinointiin liittyviä parametreja. Muutenkin jatkokehityksen kannalta olisi hyvä perehtyä tarkemmin IBM WebSphere Commercen nykyisiin sekä tuleviin REST-rajapintoihin ja niiden hyödyntämiseen joko pilvisovellusten päässä tai pilvipalvelu-widgetissä.

Pilvipalvelujen julkaisun prosessin tulisi olla selkeämpi ja helpompi: tarvittaisiin helppo tapa julkaista pilvipalvelu ja mikäli sivusto koostuisi useista pilvipalveluista tarvittaisiin selkeä tieto siitä, onko komponentti toteutettu pilvipalveluna vai tavallisena käyttöliittymäkomponenttina. Julkaisuun voisi toteuttaa automaatiota esimerkiksi siten, että palvelu julkaistaisiin versionhallinnasta testiympäristöön testien onnistuessa.

Tässä työssä toteutettu pilvipalvelu-widget toimi kuitenkin niin hyvin, että se toteutettiin oikeaan verkkokauppaprojektiin, jossa käytössä oli WCS:n versio 7. Pilvipalvelu-widget vietiin myös tuotantoympäristöön ja sillä liitettiin onnistuneesti pilvipalveluita loppukäyttäjien käytettäväksi. Jotta pilvipalvelu-widgetistä olisi hyötyä muissakin projekteissa tai muissa IBM WebSphere Commercen versioissa, täytyisi se irrottaa omaksi komponentiksi nykyisestä verkkokauppaprojektista. Tällöin pilvipalvelu-widget olisi helpompi ottaa käyttöön myös muissa verkkokauppaprojekteissa tai WCS:n versioissa.

LÄHTEET

1. Furht B. Encyclopedia of Multimedia. Springer US, 2006. s. 50–51.

2. Shklar L, Rosen R. Web Application Architecture: Principles, Protocols and Practices. John Wiley & Sons, 2009. s. 5.

3. Schuldt H. Multi-Tier Architecture. Teoksessa: Liu L, Özsu MT. Encyclopedia of Database Systems. Boston, MA: Springer US, 2009. s. 1862–1865. Saatavilla: http://dx.doi.org/10.1007/978-0-387-39940-9_652

4. Oellermann WL. Architecting Web Services. Apress, 2001.

5. IBM. WebSphere Commerce Struts framework [Internet]. Viitattu 11.10.2016. Saatavilla: https://www.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.developer.doc/concepts/csdstrutskeycompons.htm

6. Sloyer J, Microservices in Bluemix [Internet]. Viitattu 27.10.2016. Saatavilla: https://developer.ibm.com/bluemix/2015/01/19/microservices-bluemix

7. Rosen M, Lublinsky B, Smith KT, Balcer MJ. Applied SOA: Service-Oriented Architecture and Design Strategies. John Wiley & Sons, 2012. 597 s.

8. IBM. Order Management subsystem [Internet]. Viitattu 27.10.2016. Saatavilla: https://www.ibm.com/support/knowledgecenter/en/SSZLC2_7.0.0/com.ibm.commerce.developer.doc/concepts/cosordsu.htm

9. Barry DK, Dick D. Web Services, Service-Oriented Architectures, and Cloud Computing. Boston: Morgan Kaufmann, 2013. s. 15–35. Saatavilla: http://dx.doi.org /10.1016/B978-0-12-398357-2.00003-8

10. Doglio F. Pro REST API Development with Node.js. Apress, 2015.

11. Richards M. Microservices vs. Service-Oriented Architecture. O'Reilly Media, 2016. 44 s.

12. Mauro T. Microservices at Netflix: Lessons for Architectural Design [Internet]. Viitattu 11.10.2016. Saatavilla: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

13. Familiar B. Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions. Apress, 2015.

14. IBM. IBM WebSphere Commerce [Internet]. Viitattu 12.6.2016. Saatavilla: https://public.dhe.ibm.com/common/ssi/ecm/zz/en/zzs03072usen/ZZS03072USEN.PDF

15. Butterfield A, Ngondi GE. A Dictionary of Computer Science [Internet]. Oxford University Press, 2016. Saatavilla: http://dx.doi.org/10.1093/acref/9780199688975.001.0001

16. IBM. WebSphere Commerce enterprise beans [Internet]. Viitattu 12.10.2016. Saatavilla: https://www.ibm.com/support/knowledgecenter/en/SSZLC2_7.0.0/com.ibm.commerce.developer.doc/concepts/csdwcejbs.htm

17. Keith M, Schincariol M. Pro EJB 3 - Java Persistence API. Apress, 2006. s. 35–69. Saatavilla: http://dx.doi.org/10.1007/978-1-4302-0168-7_3

18. Mak G, Long J, Rubio D, Spring Recipes. Apress, 2010. s. 711–719. Saatavilla: http://dx.doi.org/10.1007%2F978-1-4302-2500-3

19. IBM. Widget architecture [Internet]. Viitattu 9.3.2016. Saatavilla: https://www.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.pagecomposerframework.doc/concepts/cpzwidgetframework.htm?lang=en

20. IBM. Management Center: E-Marketing Spots [Internet]. Viitattu 13.10.2016. Saatavilla: http://www.ibm.com/support/knowledgecenter/SSZLC2_8.0.0/com.ibm.commerce.management-center.doc/concepts/csbespotover.htm

21. RequireJS. RequireJS API [Internet]. Viitattu 5.9.2016. Saatavilla: http://requirejs.org/docs/api.html

22. IBM. Adding Management Center support for a Commerce Composer widget [Internet]. Viitattu 5.9.2016. Saatavilla: https://www.ibm.com/support/knowledgecenter/en/SSZLC2_8.0.0/com.ibm.commerce.pagecomposerframework.doc/tasks/tpzwidgetcreatecmc.htm

23. IBM. Testing your customization code [Internet]. Viitattu 5. syyskuuta 2016. Saatavilla: https://www.ibm.com/support/knowledgecenter/en/SSZLC2_7.0.0/com.ibm.commerce.pagecomposerframework.doc/tutorial/tpz_createsitewidget_test3.htm

24. IBM. Load your widget into the database by using Data Load utility [Internet]. Viitattu 5.9.2016. Saatavilla: http://www.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.pagecomposerframework.doc/tutorial/tpz_createsitewidget_dataloadwidget.htm

25. IBM. Tag: getData [Internet]. Viitattu 16.10.2016. Saatavilla: http://www.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.component-services.doc/refs/rwvgetdata.htm

26. Alcorn W, Frichot C, Orru M. The Browser Hacker’s Handbook. John & Wiley, 2014. Viitattu 19.10.2016. s. 9–10. Saatavilla: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10842312

27. IBM. WebSphere Commerce REST API for Feature Pack 8 [Internet]. Viitattu 22.10.2016. Saatavilla: http://www.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.starterstores.doc/concepts/cwvrestapi_fep8.htm

28. Shevertalov M, Mancoridis S. On the maintenance of UI-integrated mashup applications. Teoksessa: 2011 27th IEEE International Conference on Software Maintenance (ICSM). 2011. s. 203–204.

LIITE A: PILVIPALVELU-WIDGETIN DATANTARJOAJA

```
<c:if test="${!empty param.jsParams}">
   <c:set var="jsParams" value="${param.jsParams}" />
</c:if>

<c:if test="${!empty param.containerParams}">
   <c:set var="containerParams" value="${param.containerParams}" />
</c:if>

<%-- Käytä eSpotia, jos parametreja ei ole asetettu Composerissa --%>
<c:if test="${empty param.jsParams && empty param.containerParams}">
   <%@ taglib uri="http://commerce.ibm.com/coremetrics" prefix="cm" %>

   <%
      /* Hae eSpotin nimi pyynnöstä ja dekoodaa se, jos tarvetta */
      String emsName = request.getParameter("emsName");
      if (emsName != null) {
         emsName = java.net.URLDecoder.decode(emsName, "UTF-8");
        request.setAttribute("emsName", emsName);
      }
   %>

   <c:set var="emsName" value="${requestScope.emsName}"/>

   <%-- Hae tarvittavat parametrit palvelukutsulla --%>
   <wcf:getData
      type="com.ibm.commerce.marketing.facade.datatypes.MarketingSpotDataType"
      var="marketingSpotDatas"
      expressionBuilder="findByMarketingSpotName" >
      <%-- eSpotin nimi --%>
      <wcf:param name="DM_EmsName" value="${emsName}" />
   </wcf:getData>

   <wcf:eMarketingSpotCache marketingSpotData="${marketingSpotDatas}"
                            contentDependencyName="contentId" />

   <c:set var="eSpotParameters" value="" />

   <c:forEach var="marketingSpotData"
              items="${marketingSpotDatas.baseMarketingSpotActivityData}">
      <c:set var="marketingContent"
             value="${marketingSpotData.marketingContent.
                      marketingContentDescription[0].marketingText}" />

      <c:if test="${!empty marketingContent}">
         <c:set var="eSpotParameters" value="${marketingContent}" />
      </c:if>
   </c:forEach>

   <%-- Eskapoi tagit ja merkit, joita Composer saattaa lisätä --%>
   <c:if test="${fn:startsWith(eSpotParameters, '<p dir=\"ltr\">') }">
      <c:set var="eSpotParametersWithoutTags"
             value="${fn:substringAfter(eSpotParameters, '<p dir=\"ltr\">')}"/>
      <c:set var="eSpotParametersWithoutTags"
             value="${fn:substringBefore(eSpotParametersWithoutTags,'</p>')}"/>
      <c:set var="eSpotParametersWithoutTags"
             value="${fn:replace(eSpotParametersWithoutTags, '&quot;','\"')}"/>
      <c:set var="eSpotParameters" value="${eSpotParametersWithoutTags}" />
   </c:if>

   <c:set var="splittedParameters" value="${fn:split(eSpotParameters, '|')}" />
   <c:forEach var="parameter" items="${splittedParameters}">
      <c:if test="${fn:startsWith(parameter, 'jsParams')}">
```

```
63            <c:set var="jsParams"
64                   value="${fn:substringAfter(parameter, 'jsParams=')}" />
65         </c:if>
66         <c:if test="${fn:startsWith(parameter, 'containerParams')}">
67            <c:set var="containerParams"
68                value="${fn:substringAfter(parameter, 'containerParams=')}" />
69         </c:if>
70      </c:forEach>
71   </c:if>
```

Ohjelma: *CloudServiceInjector_Data.jspf-tiedosto eSpot-toiminnallisuudella*